5-18-2017

On Increasing Trust Between Developers and Automated Refactoring Tools Through Visualization

Alexander Bogart
apb1206@rit.edu

ROCHESTER INSTITUTE OF TECHNOLOGY

MASTER'S THESIS

On Increasing Trust Between Developers and Automated Refactoring Tools Through Visualization

Author:
Alexander BOGART

Supervisor:
Dr. Mohamed Wiem MKAOUER

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science in Software Engineering

in the

Department of Software Engineering
B. Thomas Golisano College of Computing and Information Sciences

May 18, 2017

The thesis "On Increasing Trust Between Developers and Automated Refactoring Tools Through Visualization" by Alexander BOGART, has been examined and approved by the following Examination Committee:

Dr. Scott Hawker

SE Graduate Program Director

Associate Professor

Dr. Daniel Krutz

Assistant Professor

Dr. Mohamed Wiem Mkaouer

Assistant Professor

Acknowledgements

Foremost, I would like to thank my adviser, Dr. Mohamed Wiem Mkaouer, for the immeasurable support he gave me during the course of this research. From the start, his constant enthusiasm encouraged me and energized me after every meeting. His advice, assistance, and expertise were invaluable to me, and without him this thesis would have surely been an impossibility.

I would like to thank Dr. Daniel Krutz, whose guidance while I was planning my thesis went above and beyond what was required; it was prompt, helpful, and crucial in ensuring that I completed my thesis at all.

I would like to thank Dr. Scott Hawker for all the advice and feedback he provided when I felt I had lost sight of my research's goal.

I would like to thank Chelsea O'Brien and the rest of the SE dept. for their prompt answers to every one of my questions, making problems disappear with a single early Monday phone call.

Finally, I would like to thank my parents for being my primary source of inspiration throughout my entire academic career, and for never once failing to remind me that they believed in me.

Rochester Institute of Technology

Abstract

Department of Software Engineering

B. Thomas Golisano College of Computing and Information Sciences

Master of Science in Software Engineering

On Increasing Trust Between Developers and Automated Refactoring Tools Through Visualization

by Alexander BOGART

In software development, maintaining good design is essential. The process of refactoring enables developers to improve this design during development without altering the program's existing behavior. However, this process can be time-consuming, introduce semantic errors, and be difficult for developers inexperienced with refactoring or unfamiliar with a given code base. Automated refactoring tools can help not only by applying these changes, but by identifying opportunities for refactoring. Yet, developers have not been quick to adopt these tools due to a lack of trust between the developer and the tool. We propose an approach in the form of a visualization to aid developers in understanding these suggested operations and increasing familiarity with automated refactoring tools. We also provide a manual validation of this approach and identify options to continue experimentation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software quality is a multi-faceted feature of any program, dependent upon design, complexity, and a myriad of other aspects. It is also an inevitability that, given enough time, changes and additions to the code base will cause its design integrity to deteriorate. Refactoring code is a necessary step to help reverse the negative effects of continuous development, short of starting from scratch every time the design deviates too far from its origins. In this manner, the design of the system can be altered without modifying its behavior or functionality.

Refactoring can be performed manually or with the aid of a tool, which can greatly increase the efficiency and accuracy of the operations. However, refactoring tools see considerably less than expected use among modern developers. Amidst various (often justified) complaints regarding these tools, many developers simply do not trust them, either in their assessments, operations, or other aspects. Developers of these tools seem to be fighting an uphill battle in trying to improve on their tools while simultaneously convincing people to use them in the first place. To aid in this endeavor, this thesis provides the following contributions:

- Firstly, an extension allowing the simultaneous visualization of multiple refactorings
- Secondly, an analysis of this extension's effect on its users' levels of trust
- Thirdly, an analysis of this extension's potential usage with regard to multiple refactoring tools

Specifically, this extension is built upon the popular refactoring tool JDeodorant [1], extending it as follows:

1. The interaction is expanded upon; we define a fine-grained visualization of multiple refactoring operations suggested to the user by the primary refactoring tool, enabling developers to better understand the impact of applying these refactorings, as well as conflicts that might arise during their application. This introduces the possibility of approximating simultaneous execution of multiple refactoring operations, rather than serially executing refactorings without a complete plan, helping to reduce the human effort required to optimize the system.

2. The ability to import refactorings is introduced; multiple refactorings operations generated by multiple refactoring tools can be parsed and imported via local files. The extension then automatically detects conflicts between these refactorings. All conflicts are conveyed as markups to the visualization, allowing the developer to easily select the respective code entities (classes, methods, fields, etc.). to identify the nature of the conflict and the appropriate refactorings to apply among the concurrent ones.

For the sake of simplicity, in this thesis, the term "extension" shall refer to the functionality we developed and added to JDeodorant, while "tool" shall refer to a fully-fledged refactoring tool, such as JDeodorant.

This thesis opens with a background in the cornerstone of refactoring: code smell detection and correction. After a review of papers on these domains, we describe the problem in detail, followed by our solution and methodology. We detail the validation performed on our approach, discuss the results and limitations, and describe avenues to not only continue this work, but experiment with new kinds of novel approaches.

Chapter 2

Background

Refactoring is a term that refers to improving the quality of a code base after design has concluded and development is underway by altering the way in which the program is structured without altering external behavior [2]. The process works by using a high-level perspective of the system to apply design-based improvements (such as relocating and restructuring fields, methods, and classes) that enhance the code's readability and modifiability. The driving force behind these changes is usually static metrics and quality attributes, including coupling, cohesion, and complexity. Many modern integrated development environments (IDEs) have commonly-used refactoring operations like Rename and Move Method refactoring as built-in functions, eliminating the need to handle any side-effects of manual refactoring that might affect the system's behavior. In other words, this functionality uses pre- and post-application checks to verify that no semantic changes were introduced.

As far as identifying which sections of the code could use refactoring, most of this work is built upon the concept of code smells, also known as design defects or anti-patterns. In general, code smells are designs that directly relate to adverse effects on software development; just as smoke indicates the presence of fire, code smells indicate the presence of "rotten" code. Static metrics can also be used to gauge overall improvements to a program's design. All in all, the refactoring process is a marriage between two domains:

1. Detection of refactoring opportunities (i.e. code smells)

2. Correction of refactoring opportunities (via refactoring methods)

In this thesis we will introduce more details regarding different types of code smells and refactoring methods relevant to this study, as well as provide an in-depth look at papers discussing detection and correction.

2.1 Code Smell Detection

In software engineering, maintaining quality is always a top priority. As development progresses and flaws inevitably begin to emerge, they generate what are known as "code smells," various indicators that code needs to be refactored or replaced [3], and can be helpful in identifying problem areas that need to be refactored. In the following section, we enumerate some of the more popular smells in literature [2], [4].

2.1.1 Code Smell Types

God Class God Classes, or God Objects, are a potential culprit of code smells. God Classes emerge when developers create a class that has too many functions, or whose functions are too large, or any other form where separation of concerns does not adequately apply. God Classes tend to break the typical laws of coupling and cohesion by controlling too much (or sometimes all) of the program, hence the name "God Class."

Data Class Data Classes are marked by their disproportionately high amount of referenced attributes compared to internal methods, effectively making them functionless data holders linked to other classes. This means that rather than providing their own methods and behavior to manipulate data, they instead merely offer data to other classes. This practice raises the static metric called efference, driving a lack of encapsulation and engendering poor data-function proximity.

Schizophrenic Class For each class, there should be no more than one key abstraction, meaning every method in a class can equally access (on average) every attribute of said class. However, it is possible to use delegation to allow one object to override another's methods, so any invocation of the delegated method will invoke the other. This means the other's context, or "self" identifier, refers to the delegator class rather than itself. This situation where "self" and "this" is ambiguous and can refer to multiple objects is referred to as "object schizophrenia" or "self schizophrenia." This lack of isolation engenders an inability to understand changes to the related classes and method invocation paths.

Refused Parent Bequest Class Refused Bequest is a code smell that affects the inheritance hierarchy. Subclasses inherit their parents' methods and data, especially those designated by the base class for use by its children with the "protected" access modifier. This relation is intended to be more significant than dependency between two unrelated classes. However, if the child does not need all the behavior provided by the parent class, it can refuse rather than request this behavior. This is an indication that the inheritance relation is mismanaged, engendering duplication and incoherent, non-cohesive interfaces.

Shotgun Surgery Method Unlike the previously mentioned code smells, Shotgun Surgery is detected at the method level, yet refers to a specific type of implementation. When implementing a feature, a developer may be required to add code to multiple locations, often with similar or slight variations, and fail to remodel the design to support these changes naturally. This engenders significant debugging problems later in development, as all of the separate locations where code was implemented will need to be updated every time the functionality goes through a change. Essentially, rather than modifying a single entity with a scalpel, the developer uses a shotgun to blow away many small portions of the code base. This defect is typically characterized by a large number of referenced classes during method execution, also known as "bottleneck operations," as they are called by and depend on many operations dispersed throughout the system.

Feature Envy Method Feature Envy is a code smell concept that applies to methods rather than classes. In broad strokes, "feature envy" means that the location of the method is ill-suited; it is more strongly related to fields and/or methods in another class or classes than its own. This breaks the convention of appropriate coupling and cohesion, as the class becomes too tightly connected and dependent on external classes, as well as containing classes that lack coherence in their functionality [5].

Message Chains Method This defect refers to when a method calls multiple data exposure methods belonging to other classes. This includes, but is not limited to, accessor methods, as data exposures can also be static methods that return an object containing part of that class. This design flaw typically manifests when a client requests an object, which requires that object to request another object, and so on and so forth. These defects can be marked during debugging by a long chain of getter-like methods, or as a sequence of temporary variables. This couples the client to this structure of navigation, meaning any change to the intermediate relationships forces the client to be changed to accommodate.

2.2 Code Smell Correction

After a code smell has been identified, the ideal way to handle it is to perform refactoring. The distinction made with refactoring is that it refers to improving the internal structure of the software without making changes to the external functionality of the code [2]. While certain performance benefits or detractions may arise as a result, the behavior of the program should remain identical. Refactorings should be visible only to the developer.

Refactoring can be applied to many types of programs, such as those written in logic [6] and

functional languages [7], [8]. However, refactoring lends itself particularly to object-oriented languages more than others as the explicit structural information available in object-oriented frameworks makes improving the design more feasible [9]. Mirroring this, object-oriented programming also relies heavily on refactoring. There's a strong emphasis in the industry to make software more reusable and modifiable [10]. Oftentimes, code needs to be re-written from scratch to accommodate these needs, but restructuring a program can often be a far more cost-effective and time-saving approach. Designing software is, after all, difficult; doubly so for reusable software. Oftentimes, a design requires many iterations before it is acceptable. Refactoring allows the design to continue to evolve and improve even after development has started. This practice directly counters the many inevitable modifications that slowly degrade the structure of a software system. These cumulative refactorings can not only negate but provide the inverse effect of software decay [11].

2.2.1 Refactoring Methods

Code smell detection can be paired with refactoring. Naturally, there is no single way to refactor a program, but popular methods have been established. Extract Class refactoring refers to taking a God Class and breaking it up into multiple, smaller classes based on the cohesion of its methods and attributes [2]. If a class has multiple features or roles, it is a good candidate for Extract Class refactoring [12]. Alternatively, Move Method refactoring takes these methods and, rather than creating a new class, moves them into existing classes where they are more cohesive [13]. On the other hand, if the issue is not the number of methods, but the size of the methods, then Extract Method refactoring is ideal, as it separates methods into external ones, replacing the previous code with a function call [14]. These are templates, and exist to be applied in the general sense.

2.3 Refactoring Tools

Thankfully, there exists a means to alleviate these downsides to refactoring in the form of refactoring tools. While there are varying degrees of automation, most refactoring tools rely on some form of user input, making them more of a semi-automated tool. A refactoring tool can include code smell detection, but this is not necessary. What differentiates detection tools from correction tools is the ability for the tool to parse or interpret source code and manipulate it in an expected way. There is typically a user interface with a selection of pre-defined operations the user can choose from, as well as the ability to preview the expected result of the operation. There must also be checks and balances in place to help ensure that only the design and not the behavior of the code is altered [15].

2.3.1 Importance & Benefits

In Martin Fowler's book *Refactoring: Improving the Design of Existing Code* (widely considered the canonical reference for refactoring), he details the four primary arguments in favor of refactoring. Firstly, refactoring has been shown to improve code design [2]. Agile methodologies that focus on changes to the design during development, and traditional software methodologies that emphasize up-front design, can both introduce substandard design. Empirical studies have shown refactoring to reduce complexity, size, cohesion, coupling, maintainability, and reusability [16]–[19]. Secondly, refactoring has been shown to make software easier to understand, especially in large programs. A Java program judged for maintainability and evolvability was discovered to have radically improved clarity and ease of use post-refactoring [20]. Thirdly, refactoring aids in the discovery of software bugs. As an indirect effect of increasing programmers' understanding of how programs work, bugs can be more easily uncovered and identified. Lastly, refactoring helps increase productivity with respect to time. Incorporating the previous three points, when a development team spends less time dealing with inadequate design, familiarizing themselves

with the code base, and performing bug detection, they reduce the overall development time. Alternatively, this time could be used to develop new features and functionality.

2.3.2 Limitations

There are, however, some drawbacks, risks, and disadvantages involved with refactoring, as with most any operation. The most constant drawback to refactoring is the time commitment needed. Refactoring operations often require many changes to be made across a given code base, and especially large systems may have a single method or variable referenced thousands of times. Performing these operations manually can be a major drain on resources, not including the time spent analyzing the program, looking for code smells and refactoring opportunities. In addition, while refactoring should not, by its very nature, alter the functionality of existing code, there is always the possibility that defects and bugs may be created in the process. Even something as innocuous as forgetting to update a reference can be a problem. A study by Kim et al. showed that over 75% of surveyed developers mentioned "regression bugs and build breaks" as a risk of refactoring, a far greater percentage than any other risk [21]. These introductions are particularly painful, as compilers will not pick up on them as readily as they will other issues, such as misspelled variable names or changes to a function's parameters. Tokuda et al. identified a number of mainstream refactoring limitations, including formatting changes, behavior not being preserved, enabling conditions not being verified automatically, and an inability to deal with preprocessor directives (though this last point is limited to specific languages) [22].

The manner in which refactoring is performed is also significant. Fowler, among others, contend for refactoring to be incorporated into the standard development cycle, used on a semi-consistent basis with occasional increases in use as issues naturally arise [2], [23], [24]. However, when developers use refactoring to try and address an issue that has already developed into a significant problem, this can result in detrimental changes to the code base, such as a negative impact on performance and a decrease in other developer's familiarity with the program [25], [26].

In general, using refactoring as a means to fix a problem rather than maintain existing quality should be treated with caution.

Drawback Mitigation

The drawbacks that plague manual refactoring, by comparison, have little impact on semi-automated refactoring. A developer need only identify to the tool the desired refactoring method and target, and the operation is performed nigh-instantaneously, sans the time necessary to recompile the project. Moreover, while the introduction of fringe-case errors are always a possibility, syntax errors are a nigh-impossibility. Thanks to the repeated iteration and improvement of refactoring definitions, explicit syntactic and semantic checks work to ensure behavior preservation. These properties are used to flag potential violations, such as incompatible signatures in member function redefinition, type-unsafe assignments, or indistinct class and member naming [9].

The time and effort saved by these tools cannot be understated. One pair estimated that restructuring two programs manually rather than with tools would have taken them days instead of hours, and weeks instead of days [22]. Another experiment showed that time spent deciphering error messages could be reduced by one third with the introduction of Refactoring Annotations, which displayed relevant information during Extract Method refactoring [27]. In general, their ability to refactor code faster while ensuring behavior is preserved makes them an ideal refactoring solution.

2.3.3 Types of Tools

As previously mentioned, refactoring tools can be classified by detection and correction. As far as correction, environments such as Eclipse and Intellij IDEA contain a number of refactoring options, such as Extract Class and Move Method, while other environments tend to, at the very least, have the ability to rename variables across the entire project. Detection tools sometimes go beyond merely identifying code smells without necessarily including correction features.

Detection tools can be defined by three levels of functionality. At the first and lowest level, most detection tools work by detecting code smells. By analyzing mostly static metrics, a tool can identify classes that would benefit from being refactored with varying degrees of accuracy [3]. Tools that solely identify metrics without analyzing them cannot really be considered refactoring tools in and of themselves. A study by Fernandes et al. of several tools' recall and precision percentages showed that, compared to a human-defined code smell reference list, certain tools identified less than ten percent of a given code smell, while others earned perfect 100% identification scores [28]. It should be noted that this does not directly compare with regards to human accuracy in code smell detection, as there is often little agreement among developers as to what constitutes a significant code smell [28]. On the same note, researcher Marinescu's metric-based detection strategy had a recall of 100% and a precision of 71% compared to God Classes identified by human subjects [29].

The second level of functionality is candidate suggestion. These tools will offer suggestions on how to refactor the identified code smells. This can be particularly useful to developers without a comprehensive understanding of the code base, as it can offer solutions the developer may not have considered, such as creating an interface for similar classes or identifying classes with high-efference couplings that would make ideal Move Method targets [30].

The third level of functionality is candidate ranking. These tools will take their candidates and, utilizing predictive algorithms, attempt to establish a confidence rating for which candidate would be the ideal implementation for the given project. Several approaches for this have been presented, such as analyzing previous changes or system complexity [31], [32]. Several approaches were implemented as JDeodorant, a refactoring tool that combines detection of a set of code smells and suggests corrections to the developer. JDeodorant will be detailed in the following section.

JDeodorant

JDeodorant is a popular refactoring tool in the form of an Eclipse plug-in [33]. It can identify a number of code smells in compilable Java programs, including God Class, Feature Envy, Long

Method, Duplicated Code, and Type Checking (also known as Switch Statement) [34]. In addition, JDeodorant automatically generates potential refactoring operations for these code smells, ranking them (when necessary) by overall impact on the code base. Certain refactorings can be visualized in a UML-style diagram, as seen in Figure 2.1. The tool ties into Eclipse's refactoring functionality, as well as providing its own, allowing the user to preview and implement a selected operation automatically, as seen in Figure 2.2. It also comes with a package view visualization, allowing the user to identify which classes in the program have the highest concentration of a particular type of code smell.

Certain refactoring tools, such as inFusion and iPlasma, utilize metric-based formulas to identify code smells [34]. In slight contrast, JDeodorant identifies God Classes by utilizing clustering to identify groupings of code entities that would improve overall system design, which is unrelated to the overall size of the class [35], [36]. While God Classes do tend to be larger in comparison to others, this is not inherently an indicator of a God Class. This clustering approach differs from the purely metric-based approach of inFusion, which utilizes class cohesion and complexity to identify God Classes [34]. For Feature Envy, the number of calls within function executions is measured as distance, and refactorings are recommended that reduce the overall "distance" within the program [37]. Its suggested refactoring opportunities are based on methods whose efference would decrease if moved to another class [33], [34].

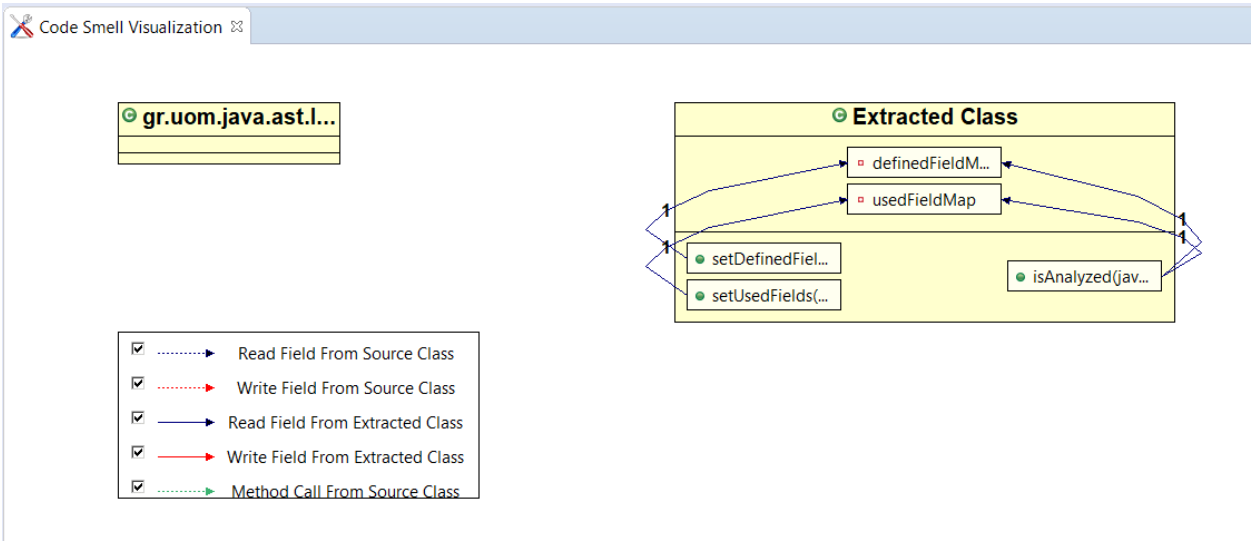


FIGURE 2.1: Visualization of an Extract Class refactoring in JDeodorant.

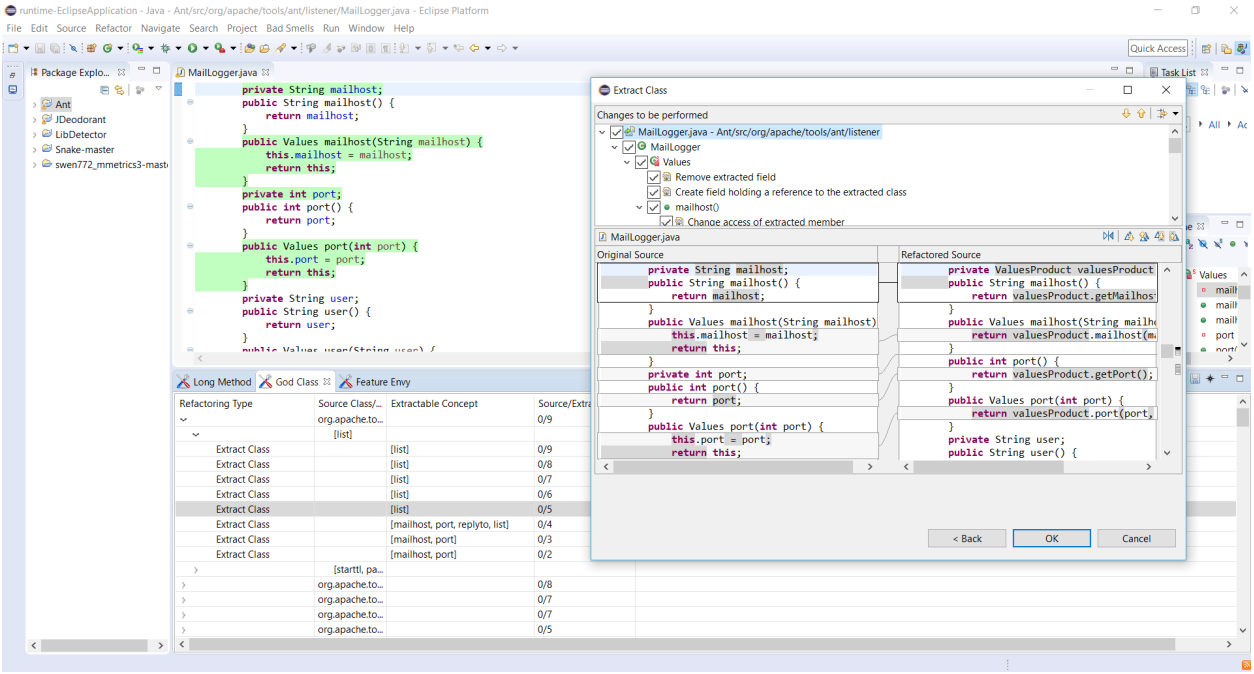


FIGURE 2.2: A refactoring in progress in JDeodorant.

Chapter 3

Literature Review

Code smell detection and correction are often analyzed in separate contexts, and often times papers will emphasize one over the other. Rather than attempting to combine both of these domains into a single literature review, this review is split into two halves, one to focus solely on code smell detection and identification, while the other focuses on the correction of code smells through refactoring.

3.1 Code Smell Detection

3.1.1 Categories

Categorizing "detection" papers is a tricky endeavor. The scope varies wildly depending on the approaches they propose. Sometimes the scope is limited to a single code smell, other times to a language, and sometimes to developers themselves, referencing human studies and human-computer interactions. If there exists a "typical" scope for detection papers, this literature review was unable to find one.

3.1.2 Analysis

While the initial plan was to categorize the code smell papers, this proved difficult in practice. Aside from being able to group the papers into those focusing primarily on detection or detectors,

the papers were decidedly unique.

The detection papers tended to be the most exploratory, focusing more on proposing novel approaches than on building solid, statistical validation of their proposals. That's not to say they were without proper procedure. For example, Vidal et al. actually built a tool that implemented their approach for prioritizing code smells for refactoring, and evaluated it in two case studies [32].

The detection papers were more varied in their topics. One identified strategies for detecting God Classes [38], another covered machine learning techniques [39], while yet another addressed the issue with conceptualization in relation to God Classes [40].

3.1.3 Personal Findings

It was significantly easier to find quality papers for the first portion of the literature review regarding refactoring methods. From a purely conjectural standpoint, this could be due to the amount of previous research done in these two domains, and the ease of acquiring statistical data. While developers may prefer to do refactoring manually rather than use an automated tool, there are still those that do utilize them and recognize their abilities. Code smell detectors, on the other hand, have an inherently more difficult job appealing to the average developer, since their detection algorithms are not immediately apparent. Their results require the developer to interpret and understand before they can develop a refactoring plan, both of which can be accomplished by manually reviewing the code. Regardless of potentially ranking the results, code smell detection also suffers from only being able to indicate defects, while refactoring can perform its function with complete accuracy. As such, the code smell papers needed to be more creative in their topics, studies, and analyses, which may have made them look less concrete when compared to the refactoring papers. However, the code smell papers held one advantage in that they managed to raise interesting questions and ideas. These papers examined unique ways of detecting code smells, such as comparing version histories, recording developer habits, analyzing vernaculars, and reflecting on the evolution of code smells.

3.2 Code Smell Correction

3.2.1 Categories

An easy way to classify most refactoring papers is by which refactoring method they focus on. Five main categories were identified: Extract Class, Extract Method, History, Move Method, and Combination. Out of the five, Extract Class, Extract Method, and Move Method are the most similar, as they are all traditional refactoring implementations. History is unique as it focuses on using version history to identify refactoring opportunities, making its papers more closely related to detection than correction. Combination papers touch on several refactoring methods.

3.2.2 Analysis

The papers centered around Extract Class and Extract Method were the most grounded. They tended to build upon the existing Extract frameworks, and proposed improvements to established approaches that seemed genuinely well-researched and tested, if occasionally somewhat situational. Move Method papers emulated this practice as well, but had the widest range of improvements of the three [13], [41]. While one paper might focus on efficiency, the other might propose a way to rank refactoring candidates. History papers had an interesting similarity in that almost all of them utilized HIST in some way, shape, or form [42], [43]. This worked in their favor, providing a common background for any reader interested in that particular domain, as well as a starting point for personal experimentation or development.

However, the Combination papers were the most unique. They benefited from not pigeonholing themselves into a single refactoring method. This lack of a singular focus did not make them too generic or large, because they tended to focus on something other than approaches to the established refactoring methods; they focused on prioritization of recommendations. One paper suggested techniques for recommending refactoring methods in large systems [44]. Another paper in particular presented an approach to prioritizing code smells for refactoring, somehow

managing to provide significant information relevant to the domain of code smells in a refactoring methods paper [32]. This was one paper that remained relevant in both literature reviews. These papers provide a valuable service in establishing common researching ground between the domains of code smell detection and refactoring methods.

3.2.3 Personal Findings

In summary, the papers reviewed thus far generally tend to focus on improvements, taking established areas of research and building upon them. Papers that presented new approaches were far less frequent. However, their topics were generally more useful to this researcher than their results. In establishing a research plan, knowing what research has been done is crucial, and seeing innovative research being done provides opportunities to either expand upon it, or branch off of it onto a new topic. The improvement papers, while generally straightforward, proved most useful in their results, showing the success and failings of their implementations, despite the fact that papers that published findings that showed their approach in a negative or even moderate light were few and far between.

Chapter 4

Problem Statement

4.1 Lack of Use

Despite the positive aspects of semi-automated refactoring, many developers continue to prefer to do refactoring manually, even when the opportunity to use a refactoring tool presents itself. In the realm of Extract Method refactoring, Kim et al. found that 58.3% of developers chose to perform their refactorings manually [21]. Another study by Negara et al. shows that even the majority of developers aware of refactoring tools and their benefits still chose to refactor manually [45]. Murphy-Hill et al. found that only 2 out of 16 students in an object-oriented programming class had used refactoring tools before [15]. A similar survey by Murphy-Hill found that 63% of surveyed individuals at an Agile methodology conference used environments with refactoring tools, and that they use the tools 68% of the time when one is available [46]. This is significant, since Agile methodologies are generally predisposed to to be in favor of refactoring, indicating the general usage must be even lower. Murphy-Hill tempers this statement by noting the likelihood of bias in the participants' responses, as well as the survey size of 112 being non-representative as it is comparatively small compared to all programmers.

Murphy-Hill also compared studies by Murphy et al. and Mäntylä et al. They show that students claim they are more likely to perform Extract Method refactoring immediately compared to Rename refactoring, yet developers are 8 times as likely to use a Rename refactoring tool than

an Extract Method refactoring tool [47]. Research by Vakilian et al. and Kim et al. also indicate that the majority of developers would prefer to apply refactorings other than Rename refactoring manually [21], [48]. There is no clear conclusion for this discrepancy, but it indicates either an underuse of Extract Method refactoring tools or overuse of Rename refactoring tools. Ultimately, it seems unrealistic to come to a concrete conclusion regarding the use of refactoring tools by all developers, but these findings show strong indirect evidence that refactoring tools are underutilized compared to their potential.

4.2 Lack of Trust

So, if refactoring tools are being underutilized, what could be the cause? There have been a number of studies and surveys done collecting information on developers' aversion to refactoring tools. Surveys by Campbell et al., Pinto et al., and Murphey-Hill all include the same barrier to entry in their findings: lack of trust [46], [49], [50]. In general, this refers to when a developer is unwilling to give control over modification of the code base to the refactoring tool due to perceived potential problems. This can manifest for a number of reasons. The developer may be unfamiliar with the tool and unwilling to risk experimenting with a tool that could modify the program in unexpected ways. The developer may be unfamiliar with the terms the tool uses, or the information it displays, or the tool may be difficult to learn or use. They may not understand exactly what the tool intends to change about their program. They may not know how the tool will affect the style or readability of the code, or they may be familiar with this and knowingly dislike what it will do to their code. Pinto et al. found that some developers will avoid suggested refactorings if they would need to trade readability for atomicity. In any of these scenarios, a more trustworthy option for the developer would be to rely on their own intuition, abilities, and experience.

Developers aso reported concerns that refactoring tools would implement poor design choices, either due to bugs in the tool, inconsistencies with the detection algorithms, or special cases with

the code base, such as reflection. Several popular refactoring tools have been shown to contain such bugs that modify program behavior without the developer ever knowing [51].

4.3 Motivation

It is this author's firm belief that the "fault" for this lack of trust lies with the tool, not the developers. Whether or not the lack of trust is justified or not is irrelevant; tools should always be built to suit the user, not the other way around. It's fundamentally impossible to change every developer's mind without first addressing the cause of the issue inherent in the tool. Therefore, the tool is where the issue must first be addressed.

The inciting incident behind this motivation came from personal experience collaborating with individuals designing a heads-up display using augmented reality hardware. One prototype feature would use GPS to identify when the wearer was in a restricted area and provide them with a visual of the shortest route out of the area. The designers soon realized that this was a terrible feature to include, for multiple reasons. Firstly, the system could be incorrect in its assessment of the area being restricted. Secondly, if the user is in a restricted area, there's likely a reason they're in there, so being told the quickest route out would be seen as a nuisance rather than a benefit. Thirdly, in an effort to not overwhelm the user with information, the fact that the user was in a restricted area was significantly less prominent than the exit path.

There's a constant tradeoff between ease of use and agency between a tool and the user. A tool that operates with a "wizard" user-interface can spare the user from having to make many complicated and potentially wrong decisions, but is forced to make two off-putting assumptions in the process:

1. The tool is correct in its assessment

2. The user does not need or want to know how the tool came to this conclusion

Tools like X-Develop [52] and Refactor! [53] use a more toolset-minded approach in their design, but in the process rely on the user to perform the precondition checks that the "wizard" tools perform automatically (albeit sometimes incorrectly due to bugs).

Many of the papers mentioned above proposed and implemented modifications to refactoring tools that addressed these concerns and barriers to entry. Campbell et al. found that utilizing preview hinting prior to performing the refactoring operations can greatly reduce apprehension regarding the tool's refactoring capabilities [49]. These hints take the form of various editor markings on the code, similar to making corrections to an essay with a red pen. However, these approaches all utilize the same medium: the written code base.

This thesis proposes an alternative approach to the issue of lack of trust in the form of a visualization diagram. Similar to the refactoring visualization generated by JDeodorant, this visualization's intended purpose is to aid the developer in easily understanding what the refactorings proposed by the tool will mean for their program. Our aspiration beyond this thesis is that developers are able to increase the level of trust between users and refactoring tools, resulting in commonplace usage of refactoring tools and an improvement in general software development.

Chapter 5

Approach

As we've established, trust is the primary concern of this thesis. Therefore, our proposed solution comes in the form of an experimental visualization whose ultimate purpose is to increase the level of trust between the developer and the refactoring tool.

5.1 Development

5.1.1 Initial Concepts

First Prototype Design

The project went through many iterations before it reached its final form. The first iteration was an empirical study. The idea was to focus on the metrics aspect of code smell identification and refactoring, and attempt to elicit information that would be useful to future developers of code smell identification and automated refactoring tools. The idea was to gather a large number of object-oriented software projects and utilize different refactoring candidate identification tools to refactor the projects until they no longer showed defects according to the tools' code smell identification algorithms. Then, static metric analysis tools would analyze snapshots of the systems before and after the refactorings, identifying established metrics such as QMOOD [54], as well as quantifiable concepts such as function complexity and class cohesion [34]. From there, it would have been a

matter of sorting through all the data and attempting to identify any interesting statistics and/or possible correlations. Potential research questions would have included:

- Which refactoring methods (out of those available) were suggested the most, and for which projects?
- Which tool's suggestions provided the best statistical improvements to the code?

However, there were several flaws with this plan from the onset. There was a misconception on this author's part regarding what constituted acceptable, or least significant, research into this domain, effectively limiting it to empirical research or human studies. Due to time restrictions a human study was impractical, so the project plan was pigeonholed. This restriction was lifted when the concept of manual evaluation was brought up during a discussion. In essence, a tool produced for the sake of research need not only be for collecting data. The process of creation can itself be a contribution, with the design and final implementation acting as a proof of concept for a previously unexplored method or idea, verifying its practical potential and opening up new avenues for continued research.

Additionally, this original approach appeared lacking in its contributions, especially with regards to the research questions, and was scrapped. Upon revisiting the idea, focus shifted from what data these questions could provide to how this information could be used, and what questions could be answered if these types of data were made available. The idea of a proof of concept and an aversion to human studies were carried over through the iterations.

Second Prototype Design

The second concept was that of a refactoring implementation view. The initial designs for this tool were very similar to the first iteration; it would take a project and, with a given set of refactorings, show how the static metrics of the system would change. This was expanded upon with the idea of including a statistic for defects introduced as a result of the given refactorings. After all, new

defects would undermine the entire refactoring process, and may prove even more detrimental since they evaded initial detection.

From there, the design began to expand to include all manner of useful views for the user. These views included graphs of various static metrics, dependency graphs, and views that would identify relevant information about refactoring opportunities, such as the last time a class was refactored, or if the refactoring had been previously undone by another developer. We began to rank the views by importance and cost to develop, with the goal of selecting a significant but manageable number of views.

This relates to the motivation of mitigating the issues of a tool that makes decisions for the user and does not inform them of how it made these decisions. Each view was designed to increase the user's awareness of the current context, allowing them to make more well-informed decisions regarding what to refactor, and the manner in which to do it.

One view that saw partial development was a metrics visualization view. Its purpose was to utilize a treemap (specifically, a 2D colored nested boxes representation [55]) or similar visualization technique to display the classes of a program and their connections, either via package location, method calls, or inheritance/dependency. It would allow the user to glance at the metrics used by the refactoring algorithm to identify code smells, granting the user a greater understanding of why the tool suggested the candidates it did, and whether these suggestions were valid or not. This is by no means an original concept; there are several fantastic implementations already, such as CodeCity [56], which utilizes a 3D city metaphor visualization technique [55], as seen in Figure 5.1.

Ultimately, we moved away from this design. Combining all these views into a single tool might have yielded dividends in helping developers understand refactoring tools and, by extension, increase their trust in the tools' suggested refactorings, but a large number of pre-existing tools would need to be combined to work in tandem, requiring significant development time merely to get them to run in the same platform. Ease of use is always a necessity, and this tool was simply too large for our scope. Therefore, we shifted our attention to using existing information to

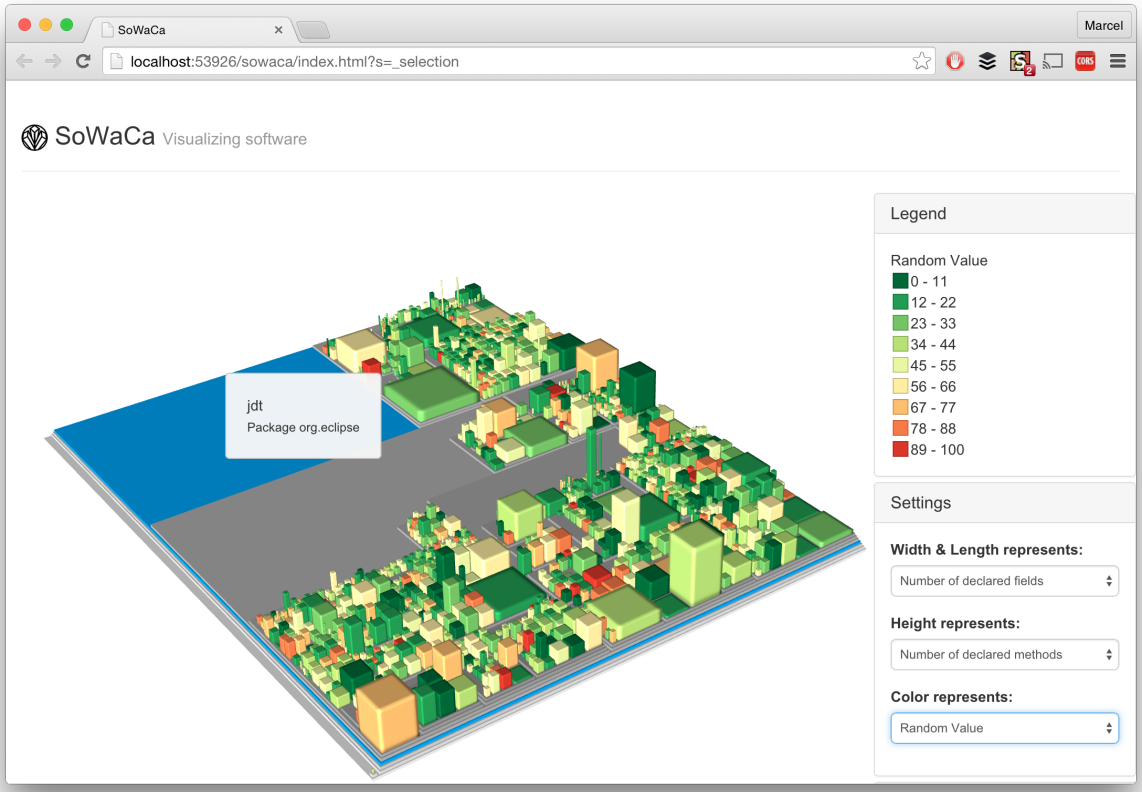


FIGURE 5.1: A screenshot of CodeCity in action [57].

create a single, unique view that could act as a supplement to an existing refactoring tool, leading to our final design and implementation.

5.1.2 Final Implemented Design

Technical Decisions

The majority of the technical design decisions were made to either keep with convention or increase ease of development. Java was chosen as the primary development language due to its purely object-oriented nature and refactoring techniques' predilections towards object-oriented

languages. Fernandes et al. found that out of 84 code smell detection tools, Java dominated, both in terms of languages that the tools analyze as well as languages the tools were developed in [28]. They also found that an equal number of the tools were plug-ins and standalone programs. With all other things being equal, rather than a standalone application, we decided to build off an existing tool, eventually deciding on the Eclipse plug-in, JDeodorant. This proved to be an ideal candidate as there are few completely open-source refactoring tools as popular as JDeodorant. Additionally, the Eclipse plug-in library is fairly substantial, and would provide an existing base for development, limiting potential issues related to building a new application from the ground up.

Visualization of Multiple Refactorings

From the start, we drew inspiration from JDeodorant's visualization feature, which creates a UML-styled view of the results of a single refactoring operation, scoped to the relevant classes (in the case of Extract Class refactoring and Move Method refactoring, this would mean the source and target classes). After considering its strengths and weaknesses with regards to conveying information to the user, we theorized that the entire refactoring operation, rendered in such detail, could be simplified to a single element in a larger visualization. Utilizing the existing code used in JDeodorant to generate diagrams, we experimented with different methods of representing this information, as well as how prominent and detailed each piece of information could be.

Developers rarely apply only a single refactoring operation to large, enterprise-level systems. Even an experienced user might have difficulty planning such a task. For a developer unfamiliar with a given program, trying to comprehend what dozens of operations would do to the program would be a daunting task, and would more than likely prompt them to either spend significant time analyzing and understanding the system, or blindly trust the assessment of an algorithm. This new visualization could display all these refactorings at once, simplifying their details into core concepts such as extraction, relocation, and merging, allowing the user to deep dive into a certain operation when necessary.

Importing Refactorings

While examining the visualization library used by JDeodorant, the idea to incorporate the metric visualization concept from our design's second iteration into JDeodorant's package visualization was introduced. Importing a file containing a program's metrics would allow the user to display these metrics in the same manner as JDeodorant, with statistically significant offenders identified by deeper gradients of red.

Initially we planned to incorporate the open-source Eclipse plug-in project Metrics 3 [58] to allow the operation to be performed automatically, but this proved unrealistic given the development time frame. While this idea was eventually cut due to its lack of relevance to the core project, the idea of importing information generated by another program led to the idea of importing refactorings generated by other programs and visualizing them using an extension to the JDeodorant plug-in.

This would also allow us to gather additional information for the second research question. From an internal standpoint, JDeodorant's refactorings could be tested for conflicts. However, with this feature, any other tool capable of exporting its refactoring candidates could be tested. Additionally, it allows for testing across multiple, separate refactoring tools.

A perhaps overlooked issue with refactoring tools is their inability to be used in conjunction with one another. By comparing the refactoring candidates generated by multiple tools, identifying the conflicts, and visualizing the combination of the valid refactorings, developers could utilize multiple tools to aid in their refactoring processes, rather than a single one. This would allow for more coverage and suggestions, and could potentially increase the user's confidence in the candidates if multiple tools' suggestions were the same.

The refactorings used in the visualization can be selected from those generated by JDeodorant (restricted to God Class and Feature Envy) so long as the project is compilable. Alternatively, refactoring operations can be imported via text files. These do not require the project to be loaded into Eclipse or compilable. The visualization can also display refactorings for multiple projects

simultaneously.

5.2 Utilization

5.2.1 Code Smell Selection

The extension to the JDeodorant plug-in can be accessed by selecting either the "God Class" or "Feature Envy" selections from the "Bad Smells" dropdown. After the project selection has been parsed and code smells have been identified, a new column labeled "Implement?" will be visible on the far right. Checking one of these boxes will add the selected operation to the visualization, as seen in Figures 5.2 and 5.3 (close-ups of these can be seen in Figures A.1 and A.2). For God Classes, selecting a parent row will also select all its children. These operations often include overlapping elements, so this is most likely useful for scouting potential refactoring combinations. The selection can be cleared using the "Clear Selected Candidates" button. Note that this is the only way to remove operations from the visualization; selecting a new project or importing a file will not reset the selection, as some users may want to visualize refactorings across multiple projects. It will also only clear God Class refactorings or Feature Envy refactorings, depending on which view is selected at the time.

5.2.2 Refactoring Visualization

The final design is centered around combining multiple refactorings into a single, presentable view. To visualize the selected refactorings, click the "Visualize Selected Candidates" button. This will open the Code Smell Visualization view, as seen in Figure 5.4. The entities in this particular diagram are classes, with the operations represented as arrows between the classes. The scope of the diagram is the directly affected classes. The number of entities (methods and fields) modified is represented on the connecting line, with the number of non-conflicting entities over the total. If a class has Extract Class refactoring performed on it multiple times, the extracted classes will be visualized in a single location for simplicity's sake. The connections will change color based

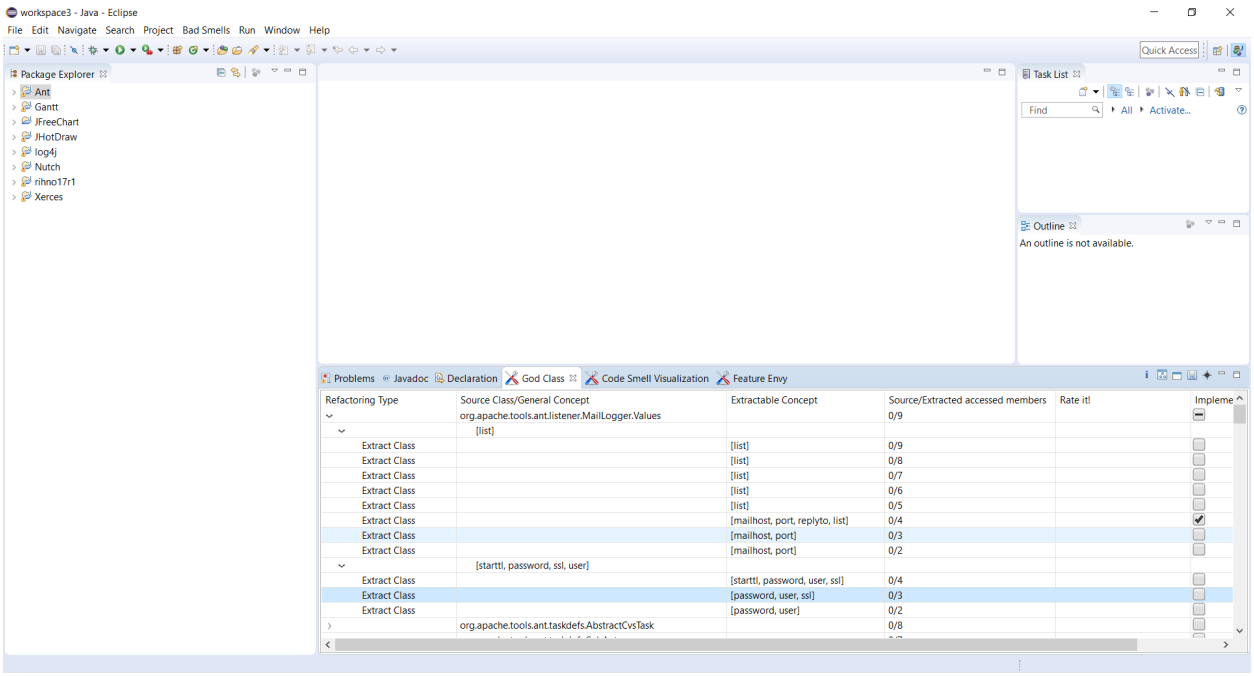


FIGURE 5.2: Selecting an Extract Class refactoring to visualize in the extension to JDeodorant.

on the number of conflicting refactorings. If there are no conflicts, the arrow will be green. Black indicates some conflicts, while red indicates over 75% of the entities involved in that class' refactorings conflict with other operations. Hovering over either the source class or the connection will cause a tooltip to appear displaying the details about the entities involved in the operations. It also displays which types of operations the conflicting entities are a part of. An example of a visualization with conflicts can be seen in Figure 5.5. This information can be copied for later use. The class diagrams are simple rectangles, and are arranged in two columns to minimize the chance that connections will overlap with another element of the diagram, though this is far from ideal.

5.2.3 Import Refactorings

Refactoring operations can also be imported by clicking the "Import Refactoring" button found in the Code Smell Visualization view, returning to either the God Class or Feature Envy view, and

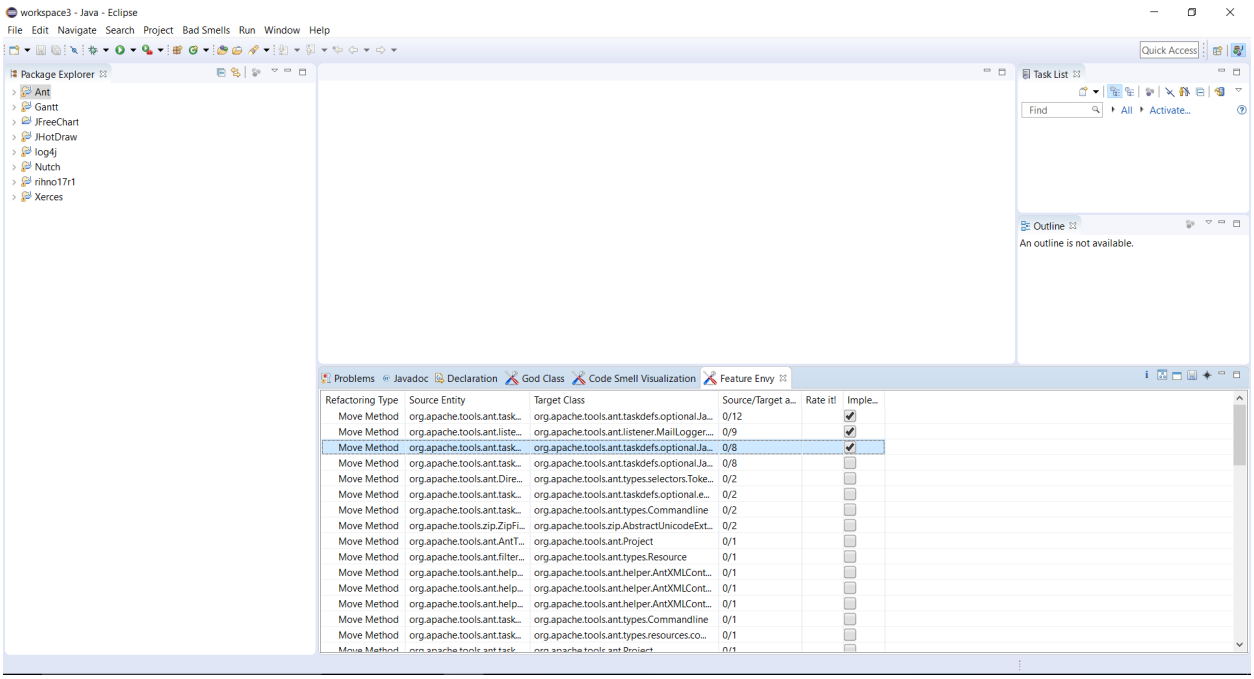


FIGURE 5.3: Selecting multiple Move Method refactorings to visualize in the extension to JDeodorant.

clicking the "Visualize Selected Candidates" button again. This will display each refactoring located in the file alongside any other selected operations. These can be cleared in the same way as the other refactorings. This feature currently parses files written in a similar format as JDeodorant's exported files, in a tab-delimited text file. We generated these from Excel files, as seen in Figure 5.6. This is also the least-developed feature in the extension to the tool, as at the time of development we lacked a specific use case for how to utilize this feature. For example, one use case suggested during development would have allowed the user to input a custom refactoring operation through a specific user interface. This feature's ultimate inclusion in the extension was primarily to receive feedback on whether its inclusion was welcome or not, as well as to perform internal testing for conflicts between refactorings generated by various tools.

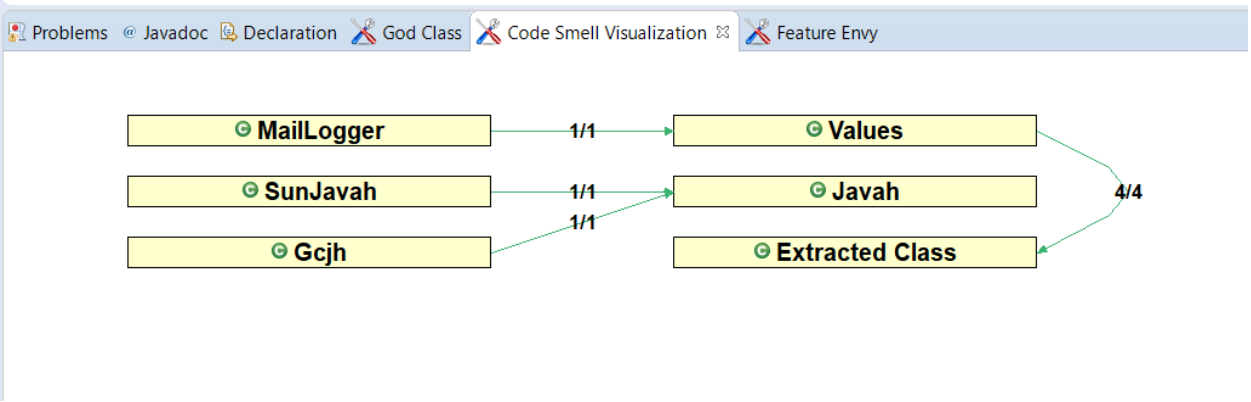


FIGURE 5.4: Visualization of the refactoring operations selected in Figures 5.2 and 5.3.

5.3 Contributions

As our goal for this work is to aid developers in becoming more familiar with automated and semi-automated refactoring, our extension will be available online to the public as an open-source project [59]. This would not have been possible without the original developers of JDeodorant making their tool open-source as well, and for that we are grateful. In addition to this extension itself, we present the design process that led to its creation as a contribution. The development of the extension itself serves as the precursor to the validation, essentially serving as a proof of concept. Whether or not the tool is successful is, naturally, secondary to expanding the body of knowledge and progressing the development of refactoring tools.

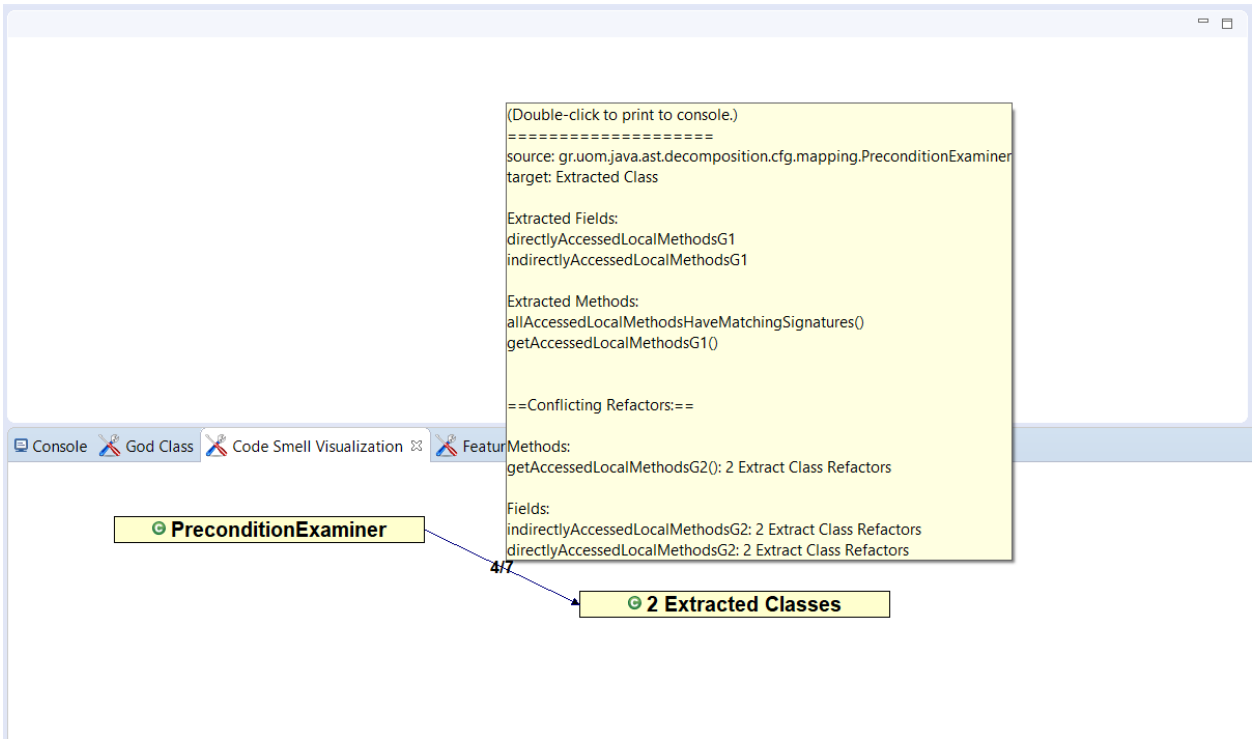
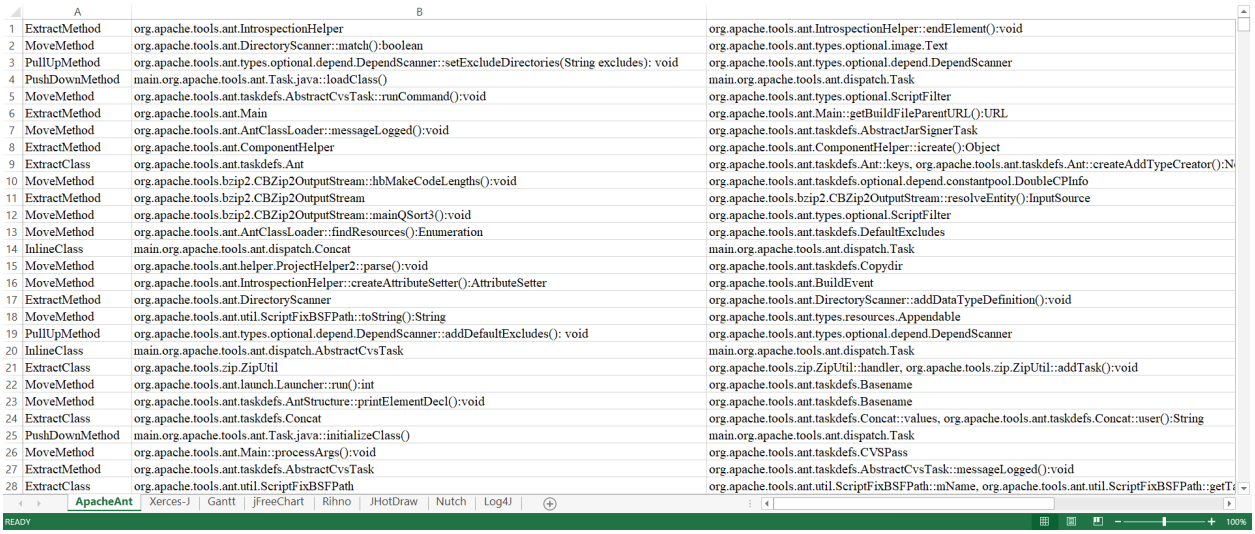


FIGURE 5.5: A visualization of conflicting Extract Class refactorings, with a tooltip.



	A	B	
1	ExtractMethod	org.apache.tools.ant.IntrospectionHelper	org.apache.tools.ant.IntrospectionHelper::endElement():void
2	MoveMethod	org.apache.tools.ant.DirectoryScanner::match():boolean	org.apache.tools.ant.types.optional.image.Text
3	PullUpMethod	org.apache.tools.ant.types.optional.depend.DependScanner::setExcludeDirectories(String excludes): void	org.apache.tools.ant.types.optional.depend.DependScanner
4	PushDownMethod	main.org.apache.tools.ant.Task.java::loadClass()	main.org.apache.tools.ant.dispatch.Task
5	MoveMethod	org.apache.tools.ant.taskdefs.AbstractCvsTask::runCommand():void	org.apache.tools.ant.types.optional.ScriptFilter
6	ExtractMethod	org.apache.tools.ant.Main	org.apache.tools.ant.Main::getBuildFileParentURL():URL
7	MoveMethod	org.apache.tools.ant.AntClassLoader::messageLogged():void	org.apache.tools.ant.taskdefs.AbstractJarSignerTask
8	ExtractMethod	org.apache.tools.ant.ComponentHelper	org.apache.tools.ant.ComponentHelper::icreate():Object
9	ExtractClass	org.apache.tools.ant.taskdefs.Ant	org.apache.tools.ant.taskdefs.Ant::keys, org.apache.tools.ant.taskdefs.Ant::createAddTypeCreator():N
10	MoveMethod	org.apache.tools.bzip2.CBZip2OutputStream::hbMakeCodeLengths():void	org.apache.tools.ant.taskdefs.optional.depend.constantpool.DoubleCPInfo
11	ExtractMethod	org.apache.tools.bzip2.CBZip2OutputStream	org.apache.tools.bzip2.CBZip2OutputStream::resolveEntity():InputSource
12	MoveMethod	org.apache.tools.bzip2.CBZip2OutputStream::mainQSort3():void	org.apache.tools.ant.types.optional.ScriptFilter
13	MoveMethod	org.apache.tools.ant.AntClassLoader::findResources():Enumeration	org.apache.tools.ant.taskdefs.DefaultExcludes
14	InlineClass	main.org.apache.tools.ant.dispatch.Concat	main.org.apache.tools.ant.dispatch.Task
15	MoveMethod	org.apache.tools.ant.helper.ProjectHelper2::parse():void	org.apache.tools.ant.taskdefs.Copydir
16	MoveMethod	org.apache.tools.ant.IntrospectionHelper::createAttributeSetter():AttributeSetter	org.apache.tools.ant.BuildEvent
17	ExtractMethod	org.apache.tools.ant.DirectoryScanner	org.apache.tools.ant.DirectoryScanner::addDataTypeDefinition():void
18	MoveMethod	org.apache.tools.ant.util.ScriptFixBSFPath::toString():String	org.apache.tools.ant.types.resources.Appendable
19	PullUpMethod	org.apache.tools.ant.types.optional.depend.DependScanner::addDefaultExcludes(): void	org.apache.tools.ant.types.optional.depend.DependScanner
20	InlineClass	main.org.apache.tools.ant.dispatch.AbstractCvsTask	main.org.apache.tools.ant.dispatch.Task
21	ExtractClass	org.apache.tools.zip.ZipUtil	org.apache.tools.zip.ZipUtil::handler, org.apache.tools.zip.ZipUtil::addTask():void
22	MoveMethod	org.apache.tools.ant.launch.Launcher::run():int	org.apache.tools.ant.taskdefs.Basename
23	MoveMethod	org.apache.tools.ant.taskdefs.AntStructure::printElementDecl():void	org.apache.tools.ant.taskdefs.Basename
24	ExtractClass	org.apache.tools.ant.taskdefs.Concat	org.apache.tools.ant.taskdefs.Concat::values, org.apache.tools.ant.taskdefs.Concat::user():String
25	PushDownMethod	main.org.apache.tools.ant.Task.java::initializeClass()	main.org.apache.tools.ant.dispatch.Task
26	MoveMethod	org.apache.tools.ant.Main::processArgs():void	org.apache.tools.ant.taskdefs.CVSPass
27	ExtractMethod	org.apache.tools.ant.taskdefs.AbstractCvsTask	org.apache.tools.ant.taskdefs.AbstractCvsTask::messageLogged():void
28	ExtractClass	org.apache.tools.ant.util.ScriptFixBSFPath	org.apache.tools.ant.util.ScriptFixBSFPath::mName, org.apache.tools.ant.util.ScriptFixBSFPath::getTa

ApacheAnt | Xerces-J | Gantt | jFreeChart | Rihno | JHotDraw | Nutch | Log4J

READY 100%

FIGURE 5.6: An Excel file containing refactoring operations to be imported.

Chapter 6

Validation

In order to evaluate our extension's abilities, we conducted a set of experiments based on eight open-source systems. In the following section, we first present our research questions and then describe and discuss the obtained results.

6.1 Research Questions

We have defined three research questions that address the applicability, the performance in comparison to existing refactoring approaches, and the usefulness of the extension. The three research questions are as follows:

6.1.1 RQ1: To what extent can our approach help the simultaneous selection and execution of multiple refactorings to developers?

The specific benefits of refactoring tools are difficult to quantify due to the unique nature with which developers refactor and utilize refactoring tools. While these can be inferred and elicited through surveys and human studies, it is inherently impossible to completely understand the inner workings of every developer, divided into groups by age, experience, and personal preferences. Therefore, if causation is out of reach, one can at the very least identify correlations. Our goal is not to prove that this approach has a statistically proven benefit for developers. Rather, we

intend to discover through concentrated evaluations and individual, written responses if there is a correlation between the use of this type of visualization and noticeable benefits to refactoring, including both time spent refactoring and the developer's willingness to use the given refactoring procedure. This serves as the first step in showing that this avenue of research may yet bear fruit.

6.1.2 RQ2: Can the use of this extension make the suggested refactorings more trustworthy in the eyes of the developer?

As the ultimate goal is to increase the level of trust between the developer and the tool, we asked a select number of developers what their impressions of the extension and its visualization were, and whether or not it helped them to better understand the refactorings proposed by JDeodorant. This particular wording was chosen due to "trustworthiness" being a difficult concept to quantify. Additionally, this provides the user with the option to directly compare two states (with and without the extension).

6.1.3 RQ3: To what extent can our approach efficiently detect conflicting refactorings provided by multiple refactoring approaches?

A reasonable cause for concern among developers is the simultaneous use of multiple refactoring tools on a single project. Despite tools being shown to have high levels of agreement in certain instances [28], this does little to mitigate the fact that a single contradiction could spell disaster for the program, limiting the refactorings' effectiveness and generating a suboptimal design at best, and introducing new bugs and defects at worst. This is an inherent issue among tools that rely on different code smell detection and/or candidate suggestion algorithms. However, one of the goals for the visualization is to convey to the developer quickly, easily, and simply any immediate conflicts that will arise from the implementing the planned suite of refactorings. Using the ability to import refactorings, we examined multiple refactoring tools' suggestions to see if there were

Project	Release	# of Classes	KLOC	# of Code Smells (identified by inFusion)
Xerces-J	v2.7.0	991	240	91
JHotDraw	v6.1	585	21	25
JFreeChart	v1.0.9	521	170	72
GanttProject	v1.10.2	245	41	49
Apache Ant	v1.8.2	1191	255	112
Rhino	v1.7R1	305	42	69
Log4J	v1.2.1	189	31	64
Nutch	v1.1	207	39	72

TABLE 6.1: Selected Projects

conflicts between refactorings both within the same tool and between separate tools, and if it might be possible in either case to create a safe refactoring suite.

6.2 Projects Under Study

To evaluate the extension, we used a set of well-known, open-source Java projects. We applied the extension to eight of these projects: Xerces-J, JHotDraw, JFreeChart, GanttProject, Apache Ant, Rhino, Log4J, and Nutch. Xerces-J is a family of software packages for parsing XML [60]. JFreeChart is a free tool for generating charts [61]. Apache Ant is a build tool and library specifically conceived for Java applications [62]. Rhino is a JavaScript interpreter and compiler written in Java and developed for the Mozilla/Firefox browser [63]. GanttProject is a cross-platform tool for project scheduling [64]. Log4J is a popular logging package for Java [65]. Nutch is an Apache project for web crawling [66]. JHotDraw is a GUI framework for drawing editors [67].

We selected these eight systems for the evaluation because they range from medium to large in size, they are open-source, they have been actively developed over the past ten years, and their development has not experienced slowdown due to their design. We used multiple projects rather than a single one to mitigate the issue of a project being easier or harder to refactor than others. Table 6.1 provides some descriptive statistics about these eight programs.

6.3 Manual Evaluation

6.3.1 Refactoring by Developers

For the eight Java projects chosen, a combined total of ten classes with potential code smells were manually identified in each project. The code smells were restricted to God Class and Feature Envy, as these were the only code smells supported by the extension at the time of writing. These projects were then grouped into pairs.

Eight developers experienced with refactoring operations participated in this evaluation. These developers remained anonymous to this author, as they were colleagues selected by Dr. Mkaouer, and all communication with them was done through him. These were all developers with experience using refactoring tools, as we desired the input of those familiar with this domain rather than those with no experience, as the extension had not been developed with that use case as its primary goal. All that was made known to this author was the number of participants, so a folder was constructed for each participant, containing a set of written instructions, a video tutorial demonstrating the extension's use, and the following requisite files. Each developer was provided with a copy of JDeodorant with the extension and two pairs of projects (four of the eight projects in total). They were each given a list of the classes containing the code smells, and asked to refactor the projects. The developers were instructed to refactor one pair of projects using the tool and visualization (extension), while the other pair were to be refactored without the aid of the visualization (see Table 6.2). The developers were asked to record how long it took to refactor each pair of projects. There were no requirements given on which projects to refactor first.

Additionally, each developer was provided with a short questionnaire. This asked for the developers to rate the features of the extension, provide feedback on its features and potential improvements and additions, and to describe how this particular extension affected both their refactoring process and opinion of/trust in the refactoring tool.

Developer	**Programs**	**Refactoring Tool**
Developer 1	JFreeChart, JHotDraw	JDeodorant
	Apache Ant, GanttProject	JDeodorant + Extension
Developer 2	Apache Ant, GanttProject	JDeodorant
	JFreeChart, JHotDraw	JDeodorant + Extension
Developer 3	Xerces-J, Rhino	JDeodorant
	Log4J, Nutch	JDeodorant + Extension
Developer 4	Log4J, Nutch	JDeodorant
	Xerces-J, Rhino	JDeodorant + Extension
Developer 5	JFreeChart, JHotDraw	JDeodorant
	Apache Ant, GanttProject	JDeodorant + Extension
Developer 6	Apache Ant, GanttProject	JDeodorant
	JFreeChart, JHotDraw	JDeodorant + Extension
Developer 7	Xerces-J, Rhino	JDeodorant
	Log4J, Nutch	JDeodorant + Extension
Developer 8	Log4J, Nutch	JDeodorant
	Xerces-J, Rhino	JDeodorant + Extension

TABLE 6.2: Distribution of Programs Among Developers

Refactoring Experiment and Questionnaire Results

Refactoring Times The refactoring experiment showed little correlation in refactoring times between developers asked to refactor the same projects with the same tools. Differences between refactoring times tended to vary between ten and 15 minutes, with extremes as low as five and as high as 29.

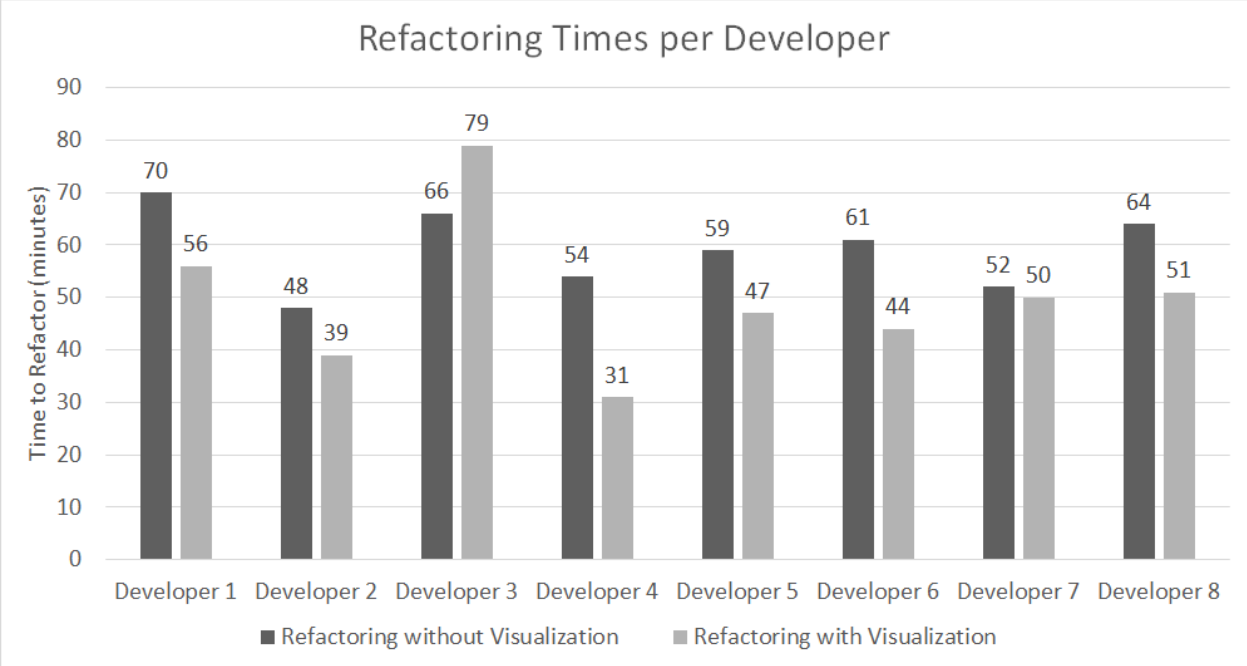


FIGURE 6.1: Refactoring Times per Developer

As shown in Figure 6.1, the total time to refactor the projects decreased when using the visualization in all but one instance. Feedback from the developers indicated universal appreciation for the visualization. Responses indicated that it helped principally with planning which refactorings to implement, as well as understanding what the changes to the system design would be. The ability to plan multiple refactorings at once, or "batch fix," seemed to help significantly with refactoring times, even when the developers needed to perform additional refactoring afterwards. One developer noted that being able to select multiple candidates alone was a benefit. While some

claimed that using the tool was easier than their static analysis methods, others acknowledged its benefits while still retaining their preference for manual refactoring, at least so far as defining refactoring operations.

Questionnaire Results To answer our **first two research questions**, we analyzed the responses from the participating developers. The developers noted that the visualization was able to identify and prevent several conflicts. However, on at least two occasions an operation caused the code to break, forcing the developer to refactor that portion manually. Expanding the functionality of the conflict detection with additional checks may help to prevent these issues.

A number of developers commented on the lack of information displayed by the visualization. In some cases the defects were not adequately described. This makes sense, as the extension only visualizes the solutions to be implemented, not the inherent problems with the classes. This made experimenting with the selected visualizations to find the right batch somewhat more time consuming and difficult than was necessary. One developer noted that this might also be mitigated prior to visualization if the extension was able to easily convey the details of the suggested refactoring candidates. This is possible at the moment with the "visualize code smell" feature (located by right-clicking on a refactoring candidate), but this process is also slow and cumbersome when dealing with dozens of potential refactorings for a single class. All in all, additional tooltip information would help increase the visualization's viability for helping developers choose from multiple refactoring candidates by adding them to the visualization first and deselecting them later.

The developers also had positive responses to the core features of the base JDeodorant tool, most notably its ability to implement the refactoring operations automatically. Comments regarding the user interface varied more widely. Some responded that they found the interface easy to use and understand, while others found it unintuitive, especially the button icons. The ability to sort the refactorings was suggested, so the user interface could be improved by implementing more features to make it easier for developers to quickly identify which refactorings they're

Feature	Developer 1	Developer 2	Developer 3	Developer 4	Developer 5	Developer 6	Developer 7	Developer 8	Average Score
Selection of multiple refactorings	3	5	3	5	3	5	5	4	4.125
Visualization of multiple refactorings	5	5	3	5	5	4	4	5	4.5
Visualization of conflicting refactorings	5	4	4	4	4	5	4	4	4.25
Import of custom refactorings	3	4	2	5	5	3	4	3	3.625

TABLE 6.3: Developer Ratings of Features (1:Bad, 5:Good)

looking for (such as a search feature).

Feature Ratings The features the subjects were asked to rate on a scale of one to five, with five being the best, were the selection of multiple refactorings, the visualization of multiple refactorings, the visualization of conflicting refactorings, and the import of custom refactorings. These results can be seen in Table 6.3. The first three all scored average ratings from between 4.125 and 4.5, while the import of custom refactorings scored significantly lower with a 3.625. This could be due to the feature not being yet fully realized, or the fact that the developers did not need to utilize this feature in their experiment, but it does indicate that this feature is not as desireable as the refactoring visualization (and its related features). This is supported by the fact that all the subjects mentioned the visualization's benefits, while none even commented on the import feature.

One of the most requested features was the ability to implement (and by extension, preview) all the selected refactoring operations at once. The developers acknowledged that visualizing their selected refactorings enabled them to refactor each one sequentially with confidence without having to run the identification feature after every application. This feature was introduced in the design phase of the extension's development, but was determined to be out of scope when we discovered there was no easy way to combine the operations into a single operation, rather

than executing them automatically in sequence, which was deemed not significant enough of an improvement to warrant the necessary development time.

6.3.2 Conflict Analysis

To answer our **third research question**, we needed to test the potential conflicts between tools' suggested refactoring candidates. We used five different approaches to identify conflicting Extract Class and Move Method refactoring operations in our eight selected projects. The five approaches are labeled as "Mkaouer" [68], "Ouni" [69], "Kessentini" [70], "O'Keeffe" [71], and "Harman" [72].

First, we identified all the refactoring operations using each approach for each project, and saved each into a CSV file, with a total of 40 files in all. After parsing the files into tab-delimited text files, we passed them into our extension, which allowed us to identify conflicting operations, even between different files, so long as the project they were refactoring was the same. Naturally, the only refactoring methods that were accepted by the extension were Extract Class and Move Method.

Table 6.4 shows how the data was recorded. Each table represents a different project, and each axis represents one of the two approaches being compared. The results in the diagonal cells (top-left to bottom-right) have the same approach name, and therefore refer to the conflicts within the approach's own operations only. Our final results are based on the percentage of conflicting refactorings, found by dividing the number of conflicting operations by the total number identified. These can be found in Appendix A.

Naturally, approaches had the lowest percentage of conflicts within their own results. The pair of approaches that produced the highest percentage of conflicts were "Mkaouer" and "Ouni." This is unsurprising, as the two had the highest internal conflict percentages. These results can be seen in Table A.9 and Figures 6.2, A.3, and A.4. Surprisingly, the two approaches that produced the highest number of operations, "O'Keeffe" and "Harman," both produced the lowest internal conflict percentages. Their conflict percentage as a pair was, comparatively, pretty low as well.

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	11				
Ouni	38	5			
Kessentini	33	28	5		
O'Keeffe	33	22	16	2	
Harman	19	12	10	10	0

TABLE 6.4: Conflicting Operations Identified Between Two Approaches for Apache Ant

Finally, we tried comparing all five approaches at once (see Table A.11). "JFreeChart" had the lowest conflict percentage while "Nutch" had the highest, with all the projects averaging out with a 28.2% conflict percentage. We were unable to find a correlation between a project's conflict percentage and any other statistic available, including the percentage of refactoring operations that were Extract Class or Move Method.

6.4 Discussion of Results

6.4.1 Number of Code Smells

While our ideal for the visualization is to be able to represent as many different types of refactoring methods as possible, in this version it currently only supports two: Extract Class refactoring and Move Method refactoring. This limits the validation that can be done with the extension, as the developer is restricted in which refactoring techniques they are allowed to use. This issue is slightly mitigated by the fact that these are two of the more versatile and popular refactoring methods, but nonetheless it remains an issue.

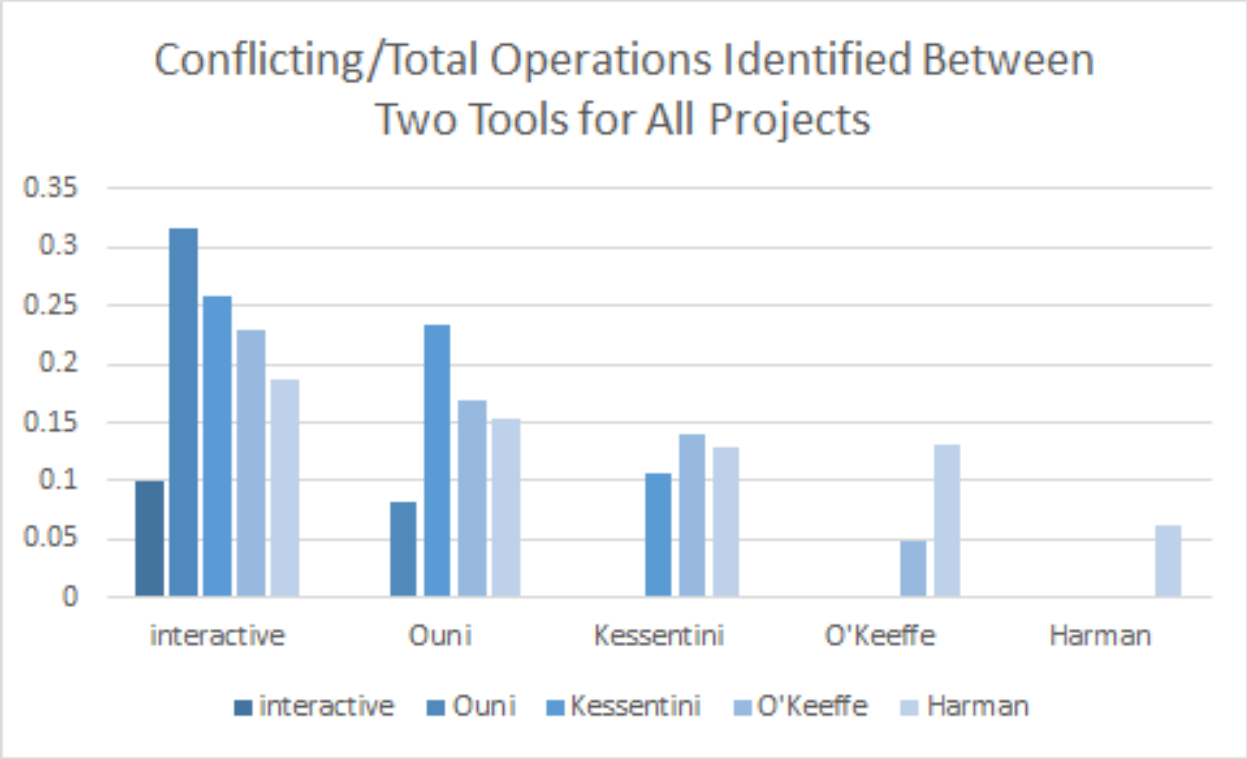


FIGURE 6.2: Bar Graph of Conflicting/Total Operations Identified Between Two Approaches for All Projects

6.4.2 Human-Computer Interaction

Additionally, the extension's design lacks input from a designer with experience in human-computer interaction, which could negatively impact its usability [73]. The primary intent behind the development of the extension was to see if the base functionality could be achieved and what results it elicited in users. The extension could undoubtedly be improved not only with expanded functionality, but with a stronger emphasis on making the extension easy to understand and use, and the visualizations clear and pleasing to the viewer. One improvement could be arranging the diagrams using a visually-appealing graphing algorithm, such as Delaunay triangulation [74], instead of a two-column grid.

6.4.3 Conflicts

While the extension can identify conflicting refactorings, this is limited to entities from a single class (methods and fields) being relocated to multiple locations. It does not identify any other defects that may be introduced to the system as a result of the refactorings, nor does it identify any new code smells that would be introduced after the refactoring operations have been performed. As a result, this feature is used as an example and test of base functionality rather than fully-fledged conflict identification.

6.4.4 Language & Tool

Naturally there are many languages aside from Java and its object-oriented cousins that utilize refactoring, and there are a multitude of development environments used for Java alone. While many aspects of the extension can be adapted for use in various environments with various languages, the technical aspects of the extension must remain limited to this specific language and environment. For example, many elements of the visualizations were immutable, such as the tooltips, and the code smell views lacked native support for checkboxes. Finding ways around these issues was not impossible, but restricted ideal and expedient development at times.

6.5 Threats to Validity

Any project built on existing work must acknowledge the source of its base of knowledge. As our extension is built upon the JDeodorant plug-in, any flaws present in that tool must also exist in our extension. This extends beyond aspects of JDeodorant limiting development of the extension's features; bugs and other defects are also a possibility. As this thesis deals with the issue of levels of trust, aspects of JDeodorant that affect this level should be taken into account when evaluating the extension's effect on trust. Without these aspects adequately defined and quantified, this remains an undefined, potential threat to validity.

As previously mentioned, there are limitations to our evaluation. Despite having eight separate developers evaluate the tool, each developer has their own preferences and experiences that affect their ability to effectively utilize certain refactoring tools. These skills can affect their ability to use the tool and its effects, which is why we assigned each developer programs to refactor with and without the extension. This should help mitigate the learning curve and fatigue threat inherent to using the extension. Additionally, each developer was provided with two different projects to mitigate the impact of a developer being particularly familiar or unfamiliar with refactoring one of the projects. Furthermore, we instituted no time limit on the refactoring or questionnaire, and provided an instructional video walking the developer through using the tool and extension to refactor a sample project.

As the crux of this thesis deals with non-quantifiable concepts, accurately interpreting human responses is a vital aspect of results analysis. Participants often have wildly varying and heterogeneous opinions. One of the ways we mitigated this threat to validity is by quantifying our results with a rating system. By having participants rate issues before responding to them, we can put together a rough indication of the developers' responses at a glance. However, this approach does not allow for concrete statistical conclusions, as developers all interpret these ratings in their own way, with no universal reference. Additionally, this still limits the number of participants we can evaluate effectively, because of the necessity of manually analyzing and interpreting each participant's responses. Ultimately, we chose the participant and project sample sizes that we did to receive clear feedback on the extension's function and impact, rather than to definitively prove its statistical effectiveness for a given demographic.

The number of participants and projects also creates a threat to validity. Because of this limit, we cannot assert that our results can be generalized and remain applicable to other projects and developers, which threatens our external validity. Accomplishing this would require future replications of this study with a larger sample size of programs and participants. Even still, trying to claim that a large enough sample size would indicate the opinions of most developers is somewhat ludicrous. Additionally, our study was limited by the extension to the use of two specific

refactoring types. While these were selected specifically to produce useful results, the quantifiable limit of two code smells could potentially hide unforeseen issues when certain combinations of refactorings are introduced.

Chapter 7

Conclusion

In this thesis, we explored the domain of code smell detection and correction. We identified the pros and cons of refactoring tools, and acknowledged a problem with modern tools: their lack of general use. We identified a potential cause of this issue to be addressed: a lack of trust between the developer and the tool, caused by limited transparency of the tool's inner workings in exchange for efficiency and simplicity. After several iterations, we proposed an approach to mitigate this issue, as well as creating an open-source implementation as an extension to an existing tool. The extension enables simultaneous visualization of multiple user-selected refactoring operations and their conflicts with one another, allowing users to better understand the impact of their application. The extension encourages continued development not only in its open-source nature, but also in its ability to utilize multiple refactoring tools through importing external files.

Analysis of the extension was handled via manual evaluation. We applied a quantitative evaluation of the extension using eight open-source systems, and a qualitative evaluation using eight developers. The feedback received shows promising results with respect to productivity and usability. Ultimately, this thesis proposes a novel approach to aid software developers in better understanding how to semi-automatically refactor their systems.

7.1 Future Work

We would be ecstatic to see work continue in this vein of increasing trust between developer and tool, even if it is not related to our visualization. New types of visualizations, or new approaches altogether, will help to increase this topic's body of knowledge and eventually lead to at least one feasible solution. While our design went through several iterations, it does not mean that aspects deemed out of our scope could not be expanded upon in future projects and shown to be viable approaches for increasing a tool's level of trust.

If development does continue on our specific visualization, we'd prioritize work on adding support for additional refactoring methods. This would help demonstrate the modularity of the system, as well as whether or not many different refactoring types over a large system can be conveyed easily. Increasing the support for different refactoring tools is also a priority. Developing either proprietary or customized ways to easily import and/or incorporate external refactoring tools would go a long way towards increasing the options available to developers with regards to choosing refactoring tools. Eventually this would ideally allow for development teams to utilize multiple refactoring tools simultaneously with little issue, increasing the list of identified code smells and refactoring candidates, as well as confidence in repeatedly suggested ones. Allowing the developer to easily self-define their own refactorings would be a significant improvement as well. Lastly, the extension's current conflict analysis only extends to fields and methods being moved and/or extracted by separate refactorings more than one time. We would like to see the extension identify deeper issues, such as defects that would or might be introduced from applying certain refactorings. These need not be fully-fledged defects, but introductions of new code smells, such as the introduction of too many small classes and an increase in efference.

Ideally, development would continue in the form of different visualization techniques, and experiments into which combinations of visualizations and other features help increase the developer's level of trust in the tool, without sacrificing the efficiency it provides.

In the end, any work that leads towards increasing the general practice and use of refactoring tools will be the ultimate success.

Appendix A

Appendix

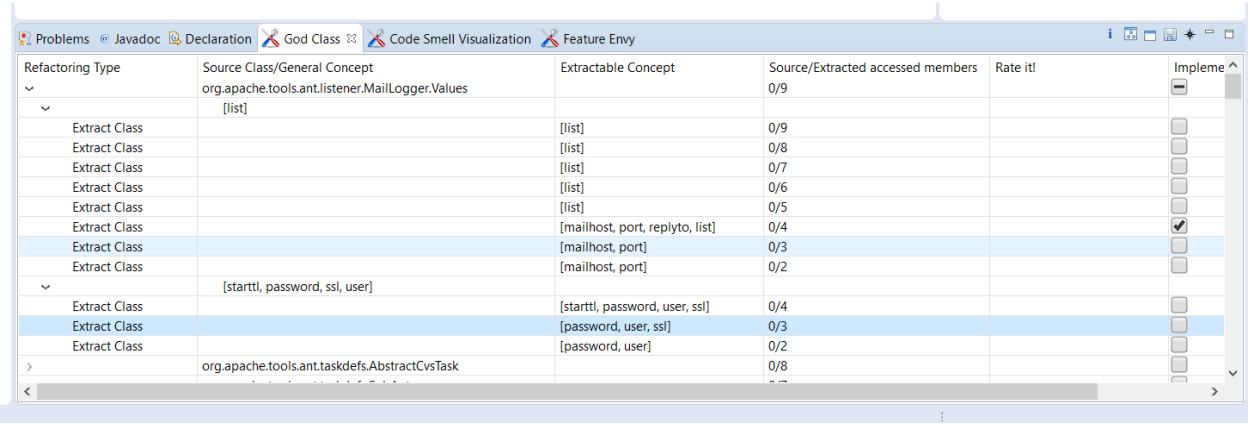


FIGURE A.1: Close-up of Figure 5.2.

	Mkaouer	Ouni	Kessentini	O'Keeffe	Harman
Mkaouer	0.268293				
Ouni	0.493506	0.138889			
Kessentini	0.445946	0.405797	0.151515		
O'Keeffe	0.375	0.26506	0.2	0.042553	
Harman	0.223529	0.15	0.12987	0.10989	0

TABLE A.1: Conflicting/Total Operations Identified Between Two Approaches for Apache Ant

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0.115385				
Ouni	0.333333	0.136364			
Kessentini	0.316327	0.3	0.086957		
O'Keeffe	0.239583	0.215909	0.2	0.045455	
Harman	0.193878	0.211111	0.173913	0.066667	0.043478

TABLE A.2: Conflicting/Total Operations Identified Between Two Approaches for Xerces-J

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0.086957				
Ouni	0.261905	0			
Kessentini	0.064516	0.023529	0		
O'Keeffe	0.178947	0.022989	0	0	
Harman	0.234043	0.116279	0	0.123711	0

TABLE A.3: Conflicting/Total Operations Identified Between Two Approaches for GanttProject

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0				
Ouni	0.116505	0			
Kessentini	0	0.021505	0		
O'Keeffe	0.033058	0.057692	0	0	
Harman	0.119658	0.06	0.018692	0	0

TABLE A.4: Conflicting/Total Operations Identified Between Two Approaches for JFreeChart

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0				
Ouni	0.378788	0.294118			
Kessentini	0.264706	0.271429	0		
O'Keeffe	0.09375	0.227273	0.058824	0	
Harman	0	0.138889	0	0.085714	0

TABLE A.5: Conflicting/Total Operations Identified Between Two Approaches for Rhino

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0.142857				
Ouni	0.482759	0			
Kessentini	0.515152	0.441176	0.526316		
O'Keeffe	0.666667	0.428571	0.53125	0.307692	
Harman	0.451613	0.375	0.527778	0.466667	0.352941

TABLE A.6: Conflicting/Total Operations Identified Between Two Approaches for Nutch

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0.105263				
Ouni	0.285714	0.086957			
Kessentini	0.292683	0.311111	0.090909		
O'Keeffe	0.15	0.136364	0.139535	0	
Harman	0.2	0.181818	0.186047	0.190476	0.095238

TABLE A.7: Conflicting/Total Operations Identified Between Two Approaches for Log4J

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0.074074				
Ouni	0.178571	0			
Kessentini	0.175439	0.101695	0		
O'Keeffe	0.096774	0	0	0	
Harman	0.066667	0	0	0	0

TABLE A.8: Conflicting/Total Operations Identified Between Two Approaches for JHotDraw

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Mkaouer	0.099104				
Ouni	0.316385	0.082041			
Kessentini	0.259346	0.23453	0.106962		
O'Keeffe	0.229222	0.169232	0.141201	0.049463	
Harman	0.186173	0.154137	0.129537	0.130391	0.061457

TABLE A.9: Conflicting/Total Operations Identified Between Two Approaches for All Projects

	Mkaouer	**Ouni**	**Kessentini**	**O'Keeffe**	**Harman**
Total Conflicting Operations Identified in All Projects	27	23	21	8	10
Total Operations Identified in All Projects	291	262	283	302	304
Conflicting/Total Operations Identified in All Projects	0.093	0.088	0.074	0.026	0.033

TABLE A.10: Statistics for All Projects per Approach

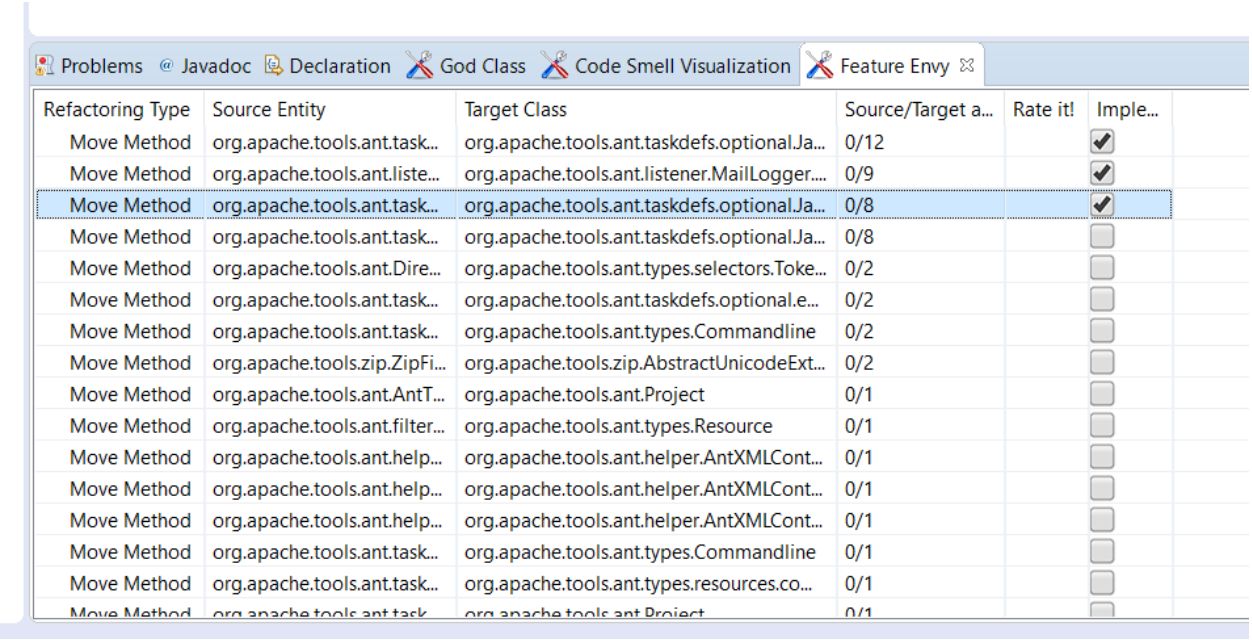


FIGURE A.2: Close-up of Figure 5.3.

	Ant	Xerces	Gantt	JFreeChart	Rhino	Nutch	Log4J	JHotDraw
Conflicting Operations Identified by All Approaches	73	71	49	31	45	44	32	21
Total Operations Identified by All Approaches	201	232	228	271	172	78	106	154
Conflicting/Total Operations Identified by All Approaches	0.363	0.306	0.215	0.114	0.262	0.564	0.302	0.136
Extract Class/ Total Operations	0.348	0.336	0.355	0.303	0.337	0.308	0.321	0.409
Move Method/ Total Operations	0.652	0.664	0.645	0.697	0.663	0.692	0.679	0.591

TABLE A.11: Statistics for All Approaches per Project

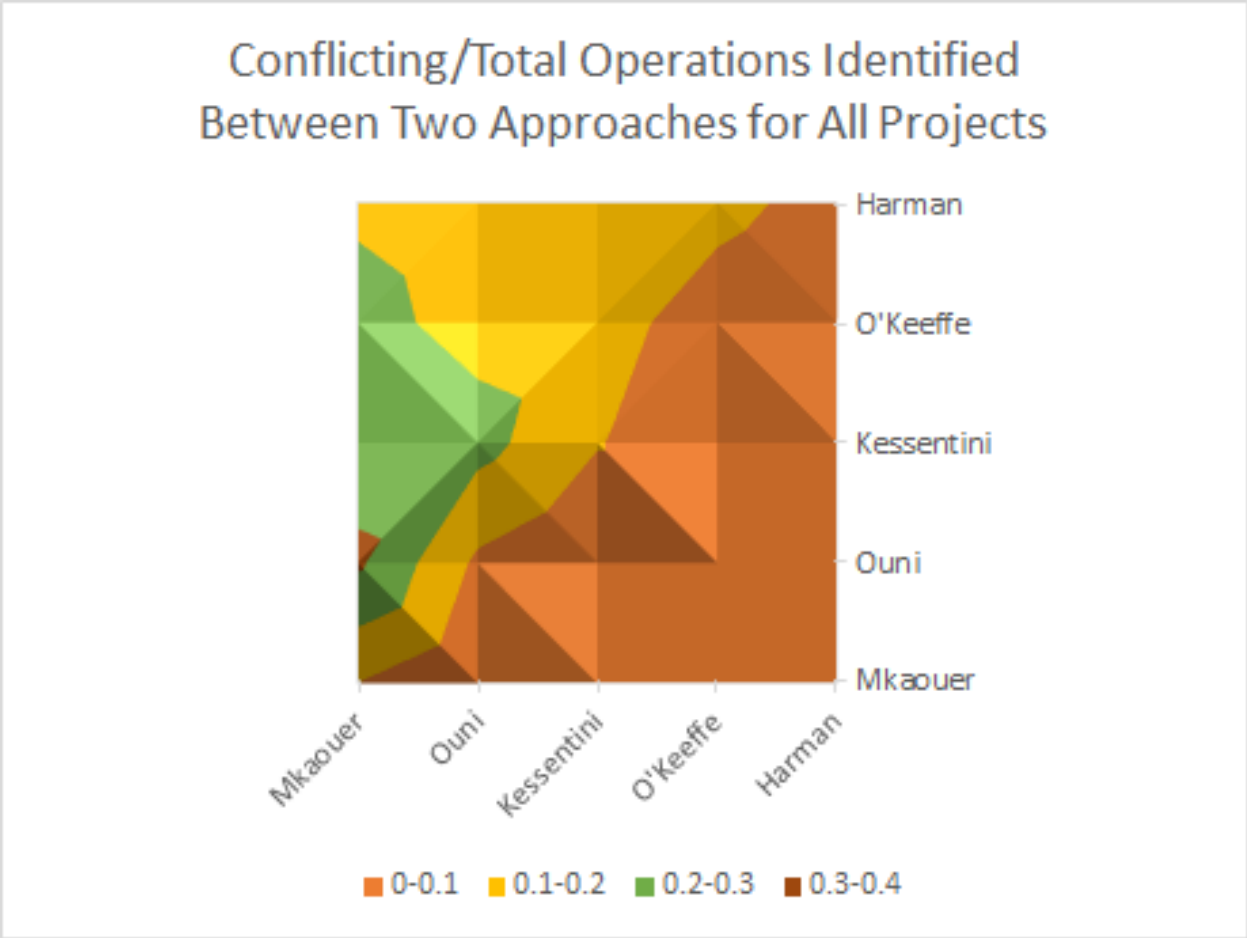


FIGURE A.3: Contour Graph of Conflicting/Total Operations Identified Between Two Approaches for All Projects

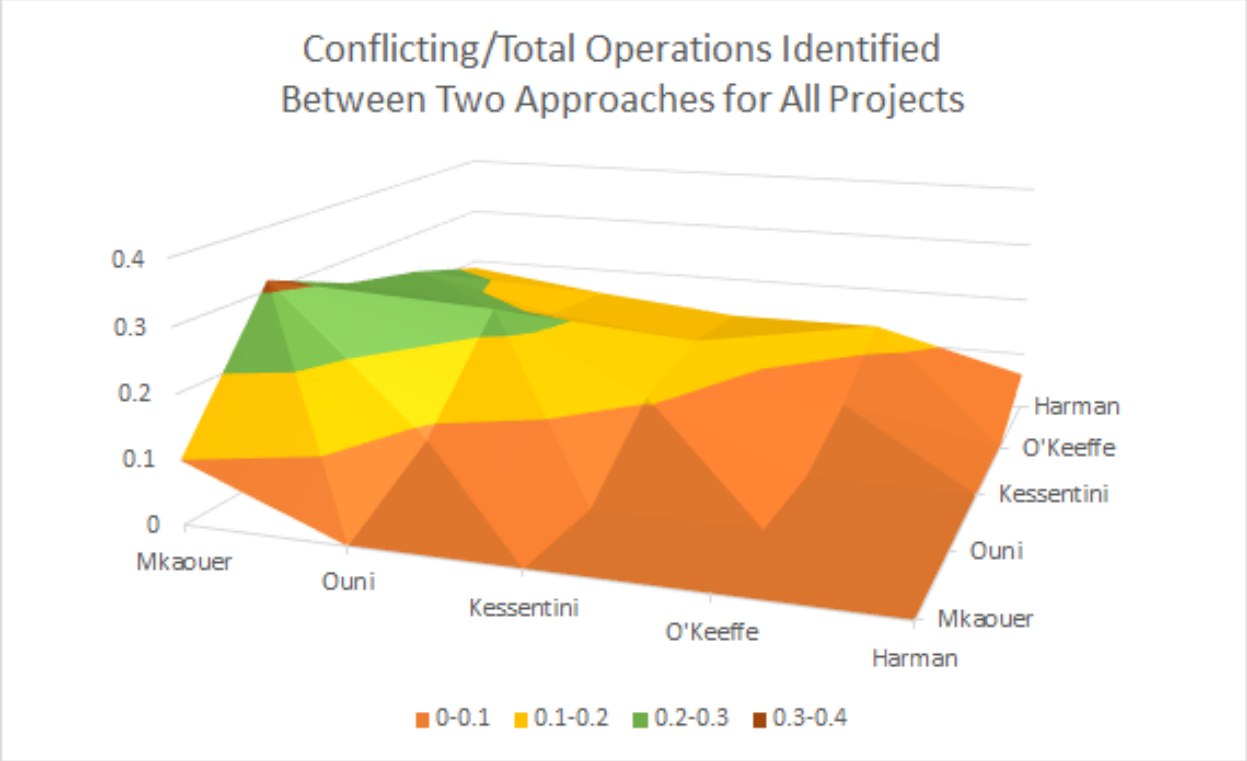


FIGURE A.4: Surface Graph of Conflicting/Total Operations Identified Between Two Approaches for All Projects

References

[1] A. Hamid, M. Ilyas, M. Hummayun, and A. Nawaz, "A comparative study on code smell detection tools", *International Journal of Advanced Science and Technology*, vol. 60, pp. 25–32, 2013.

[2] M. Fowler and K. Beck, *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 1999.

[3] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells", in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, IEEE, 2002, pp. 97–106.

[4] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[5] K. Nongpong, "Feature envy factor: A metric for automatic feature envy detection", in *Knowledge and Smart Technology (KST), 2015 7th International Conference on*, IEEE, 2015, pp. 7–12.

[6] T. Mens and T. Tourwé, "A survey of software refactoring", *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[7] H. Li, C. Reinke, and S. Thompson, "Tool support for refactoring functional programs", in *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, ACM, 2003, pp. 27–38.

[8] S. Thompson, "Refactoring functional programs", in *International School on Advanced Functional Programming*, Springer, 2004, pp. 331–357.

[9] W. F. Opdyke, "Refactoring object-oriented frameworks", PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[10] R. E. Johnson and B. Foote, "Designing reusable classes", *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[11] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data", *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[12] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: An improved method and its evaluation", *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.

[13] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models", *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2014.

[14] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings", in *Proceedings of the 22nd International Conference on Program Comprehension*, ACM, 2014, pp. 146–156.

[15] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose", *IEEE software*, vol. 25, no. 5, 2008.

[16] J. Benn, C. Constantinides, H. K. Padda, K. H. Pedersen, F. Rioux, and X. Ye, "Reasoning on software quality improvement with aspect-oriented refactoring: A case study", in *Proceedings of the IASTED International Conference on Software Engineering and Applications*, 2005, pp. 309–315.

[17] B. Geppert and F Rossler, "Effects of refactoring legacy protocol implementations: A case study", in *Software Metrics, 2004. Proceedings. 10th International Symposium on*, IEEE, 2004, pp. 14–25.

[18] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A case study in refactoring a legacy component for reuse in a product line", in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, IEEE, 2005, pp. 369–378.

[19] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?", in *International Conference on Software Reuse*, Springer, 2006, pp. 287–297.

[20] J. Ratzinger, M. Fischer, and H. Gall, *Improving evolvability through refactoring*, 4. ACM, 2005, vol. 30.

[21] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits", in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ACM, 2012, p. 50.

[22] L. Tokuda and D. Batory, "Evolving object-oriented designs with refactorings", in *Automated Software Engineering, 1999. 14th IEEE International Conference on.*, IEEE, 1999, pp. 174–181.

[23] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting refactoring activities using histories of program modification", *IEICE transactions on information and systems*, vol. 89, no. 4, pp. 1403–1412, 2006.

[24] C. Parnin and C. Görg, "Lightweight visualizations for inspecting code smells", in *Proceedings of the 2006 ACM symposium on Software visualization*, ACM, 2006, pp. 171–172.

[25] M. Pizka *et al.*, "Straightening spaghetti-code with refactoring?", in *Software Engineering Research and Practice*, 2004, pp. 846–852.

[26] F. Bourquin and R. K. Keller, "High-impact refactoring based on architecture violations", in *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, IEEE, 2007, pp. 149–158.

[27] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring: Observations and tools for extract method", in *Proceedings of the 30th international conference on Software engineering*, ACM, 2008, pp. 421–430.

[28] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools", in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ACM, 2016, p. 18.

[29] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws", in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, IEEE, 2004, pp. 350–359.

[30] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold, "Automated support for program refactoring using invariants", in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, IEEE Computer Society, 2001, p. 736.

[31] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility", in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, IEEE, 2011, pp. 25–34.

[32] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring", *Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2016.

[33] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods", *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[34] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment.", *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[35] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: Identification and application of extract class refactorings", in *Proceedings of the 33rd International Conference on Software Engineering*, ACM, 2011, pp. 1037–1039.

[36] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique", in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, IEEE, 2009, pp. 93–101.

[37] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells", in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, IEEE, 2007, pp. 519–520.

[38] J. A. M. Santos and M. G. Mendonça, "Identifying strategies on god class detection in two controlled experiments.", in *SEKE*, 2014, pp. 244–249.

[39] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection", *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[40] J. A. M. Santos, M. G. de Mendonça, C. P. Dos Santos, and R. L. Novais, "The problem of conceptualization in god class detection: Agreement, strategies and decision drivers", *Journal of Software Engineering Research and Development*, vol. 2, no. 1, p. 11, 2014.

[41] A.-R. Han, D.-H. Bae, and S. Cha, "An efficient approach to identify multiple and independent move method refactoring candidates", *Information and Software Technology*, vol. 59, pp. 53–66, 2015.

[42] D Rapu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection", in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, IEEE, 2004, pp. 223–232.

[43] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells", *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[44] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems", in *Recommendation Systems in Software Engineering*, Springer, 2014, pp. 387–419.

[45] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings", in *European Conference on Object-Oriented Programming*, Springer, 2013, pp. 552–576.

[46] E. Murphy-Hill, "Programmer friendly refactoring tools", 2009.

[47] M. V. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions", in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ACM, 2006, pp. 297–306.

[48] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings", in *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, 2012, pp. 233–243.

[49] D. Campbell and M. Miller, "Designing refactoring tools for developers", in *Proceedings of the 2nd Workshop on Refactoring Tools*, ACM, 2008, p. 9.

[50] G. H. Pinto and F. Kamei, "What programmers say about refactoring tools?: An empirical investigation of stack overflow", in *Proceedings of the 2013 ACM workshop on Workshop on refactoring tools*, ACM, 2013, pp. 33–36.

[51] M. Verbaere, R. Ettinger, and O. De Moor, "Jungl: A scripting language for refactoring", in *Proceedings of the 28th international conference on Software engineering*, ACM, 2006, pp. 172–181.

[52] *X-develop*. [Online]. Available: `http://freecode.com/projects/xdevelop`.

[53] D. E. Inc., *Coderush*. [Online]. Available: `https://www.devexpress.com/products/coderush/refactor_pro.xml`.

[54] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment", *IEEE Transactions on software engineering*, vol. 28, no. 1, pp. 4–17, 2002.

[55] P. Caserta and O. Zendra, "Visualization of the static aspects of software: A survey", *IEEE transactions on visualization and computer graphics*, vol. 17, no. 7, pp. 913–933, 2011.

[56] R. Wettel and M. Lanza, "Visualizing software systems as cities", in *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, IEEE, 2007, pp. 92–99.

[57] *Codecity*. [Online]. Available: https://marketplace.eclipse.org/content/codecity.

[58] Leonardobsjr, *Leonardobsjr/metrics3*, 2015. [Online]. Available: https://github.com/leonardobsjr/metrics3.

[59] Hakimakitak, *Hakimakitak/jdeodorantrefactoringviews*, 2017. [Online]. Available: https://github.com/Hakimakitak/JDeodorantRefactoringViews.

[60] *Xerces java parser readme*. [Online]. Available: http://xml.apache.org/xerces-j/index.html.

[61] *Jfreechart*. [Online]. Available: http://www.jfree.org/jfreechart/.

[62] C. MacNeill and S. Bodewig, *Welcome*. [Online]. Available: http://ant.apache.org/.

[63] *Rhino*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino.

[64] .

[65] *Apache log4j 2*. [Online]. Available: https://logging.apache.org/log4j/2.x/.

[66] dev@Nutch.apache.org, *Highly extensible, highly scalable web crawler*. [Online]. Available: http://nutch.apache.org/.

[67] *Jhotdraw as open-source project*. [Online]. Available: http://www.jhotdraw.org/.

[68] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation system for software refactoring using innovization and interactive dynamic optimization", in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, ACM, 2014, pp. 331–336.

[69] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-criteria code refactoring using search-based software engineering: An industrial case study", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 23, 2016.

[70] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example", in *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, IEEE, 2011, pp. 81–90.

[71] M. O'Keeffe and M. O. Cinnéide, "Search-based refactoring for software maintenance", *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[72] M. Harman and L. Tratt, "Pareto optimal search based refactoring at the design level", in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ACM, 2007, pp. 1106–1113.

[73] A. Dix, *Human-computer interaction*. Springer, 2009.

[74] D.-T. Lee and B. J. Schachter, "Two algorithms for constructing a delaunay triangulation", *International Journal of Computer & Information Sciences*, vol. 9, no. 3, pp. 219–242, 1980.